An Energy Efficient FPGA Hardware Architecture for the Acceleration of OpenCV Object Detection

by

Braiden Brousseau

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Braiden Brousseau

Abstract

An Energy Efficient FPGA Hardware Architecture for the Acceleration of OpenCV Object Detection

Braiden Brousseau

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2012

The use of Computer Vision in programmable mobile devices could lead to novel and creative applications. However, the computational demands of Computer Vision are ill-suited to low performance mobile processors. Also the evolving algorithms, due to active research in this field, are ill-suited to dedicated digital circuits. This thesis proposes the inclusion of an FPGA co-processor in smartphones as a means of efficiently computing tasks such as Computer Vision. An open source object detection algorithm is run on a mobile device and implemented on an FPGA to motivate this proposal. Our hardware implementation presents a novel memory architecture and a SIMD processing style that achieves both high performance and energy efficiency. The FPGA implementation outperforms a mobile device by 59 times while being 13.5 times more energy efficient.

Acknowledgements

I would like to thank my supervisor Jonathan Rose for his encouragement to pursue graduate studies, and as well for the many great opportunities and my experiences resulting from being his graduate student. I would like to also express gratitude to my mother Wendy Brousseau as well as my aunt Robin Norris and uncle David Halls for their support and encouragement throughout my degree. Finally, I would like to thank Jason Luu, Xander Chin, Alex Rodinov, and many other students in the lab. Their feedback and insight is appreciated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Smartphones had largely remained niche products until 2007 when Apple brought mobile computing and the paradigm of the user programmable application-centric smartphone to the masses with the launch of the iPhone [17]. Fuelled by expanding wireless data infrastructure and maturing mobile system on chips (SoC) the iPhone operating system became the first of several development-friendly modern mobile operating systems to launch in the past 4 years.

The collection, categorization and distribution of information is the driving force behind many remarkable information technology systems that have dramatically changed the structure of society. Smartphones are emerging as one of these technologies by enabling users to capture what was once an ephemeral stream of our personal senses. This data has wonderfully emergent properties when analyzed in aggregate over time and when it is shared between many people. This is made possible by our smartphones' ability to sense the world around us.

A modern smartphone knows where you are via GPS, and can see what you see with a high resolution camera; it knows how you look with a front facing camera, can hear

what you hear with a microphone, knows its orientation using a gyroscope, and knows how its moving with an accelerometer. It can communicate, compare, and interact with anything else out on the internet using wireless technology. Each sensor brings with it an additional axis to the design space of personal computing. The number of sensors is only likely to increase moving forward, bringing even more opportunity for invention.

It quickly becomes apparent, however, that only a portion of this space may be truly accessible on smartphones as they exist today. Smartphones are incredibly constrained devices particularly by energy consumption. Battery energy density is doubling only every 10 years [4], and when combined with the physical size limitations of a portable device this creates a relatively static energy budget to which all new devices, and their software, must adhere.

As a result of these energy constraints, the CPUs that power modern smartphones are optimized for power efficiency rather than raw speed and so have significantly lower performance than conventional desktop or laptop processors. For this reason, applications that require very high computation costs will remain challenging to implement on mobile devices.

One such challenging application type is in the field of Computer Vision which seeks to automatically extract information from images. Computer Vision research seeks to give machines the ability to perform visual tasks such as tracking where an object moves or classifying objects with large visual design variability into categories. These tasks are are easily performed by humans, but have thus far been very difficult for computers.

Computer Vision is important for two reasons: firstly it lends itself to many interesting mobile applications that can analyze and understand what a user is seeing, but secondly because the camera is currently the highest bandwidth sensor on most devices. If we can understand the feasibility of Computer Vision in a mobile context, it will give some sense of the feasibility of future high bandwidth and high computation application domains.

Computer Vision is a very active field of research that is exploring many different

methods to achieve the same ends. As a result there are many new algorithms and computational techniques that are being developed or applied to the field of Computer Vision regularly. In an effort to aggregate and make available this large number of complex algorithms an open source effort called the Open Computer Vision Project (OpenCV) was founded by Intel Corporation in 2006 [15]. Here many stable and fundamental vision algorithms have been made available in Open Source code with optimizations for Intel processors.

Computer Vision as a field began with attempts to have computers identify simple objects. Object detection is just one of the many tasks OpenCV can perform and it has always been one of the key capabilities of Computer Vision. OpenCV implements a well known and powerful object detection algorithm called Multi-Scale Haar Cascade Detection (MSHCD). This algorithm will be used in this work to explore Computer Vision in a mobile context and the OpenCV implementation will be used as the gold standard. An implementation of the MSHCD algorithm will be called OpenCV *compliant* if, given the same stimuli, the implementation and OpenCV produce identical results.

There are currently three main computational media on a modern smartphone, none of which are likely to scale favourably and meet the growing demand for high bandwidth digital signal processing. The most energy efficient of these methods is the use of application specific integrated circuits (ASICs). Adding an extra chip to a device, or extra hardware on an existing chip is an expensive proposition when it only performs a small number of very specific computations. This means it is done only for computations for which a large percentage of the end users will take advantage, such as video decoding. ASICs are impractical when the algorithms being computed are constantly changing and improving. This is often the case in mobile development as new creative combinations of available mobile sensor data are used to creating interesting applications. Next is the general purpose processor, but processor architecture is a mature field and the energy reduction associated with process scaling is not as pronounced as it once was. This makes

it less likely that mobile processors will achieve exponential improvements in performance per unit energy. Finally, data can be sent over the wireless network to computers in the cloud. Using the wireless radio is one of the largest energy consumer of a modern devices and overuse can quickly drain the battery, depending on the volume of data.

One way that we might increase the energy efficiency of these devices, especially in the area of digital signal processing, is the inclusion of a Field-Programmable Gate Array (FPGA) co-processor. An FPGA is a programmable digital integrated circuit. It benefits from the power efficiency associated with making a dedicated digital circuit that performs only one specific task but because it is reprogrammable it can be used for many applications. The addition of an FPGA along side a smartphone's other computational media could unlock more of this burgeoning design space and lead to more creative, interesting, and impactful uses of the stream of information being provided by modern smartphones.

1.2 Research Goals

The goal of this research is to explore the benefits of an FPGA co-processor in a mobile environment for the purpose of object detection. The specific goals are to:

1. Create a framework for utilizing existing OpenCV libraries on a mobile device;

2. Design a scalable FPGA hardware implementation of an OpenCV compliant Multi-Scale Haar Cascade Detection algorithm; and

3. Measure and compare the frame rate and energy usage of the MSHCD hardware system with alternate computational media.

1.3 Thesis Overview

Chapter 2 provides background on the state of the mobile computing landscape including the hardware and operating system platform we chose. It also provides a more in depth introduction to the OpenCV computer vision software library, a detailed overview of the algorithm used from this package and an explanation of how this algorithm has been architected on FPGAs in previous work. Chapter 3 discusses the hardware design that implements the OpenCV Multi-Scale Haar Cascade Detection algorithm, including the specific architectural enhancements that distinguishes this implementation from previous work. Chapter 4 presents the experimental methodology and gives the experimental results that measure energy consumption and overall performance. The final chapter offers a summary of the thesis as well as setting out potential future work.

Chapter 2

Background

This Chapter will examine background material relevant to the subject of this thesis. It begins by briefly outlining the mobile platforms available at the onset of this work. We then provide relevant technical details of the mobile platform used in this research, and the external FPGA used. We will give an overview of the field of computer vision and provide details about OpenCV. We describe in detail the MSHCD algorithm that was implemented in hardware as part of this work. Finally we will outline previous work on a hardware architecture of the most performance-critical section of this algorithm.

2.1 Mobile Platform Choices

When this work began in the middle of 2010, the choice of smartphone platform was not obvious. At the time there were three major platforms to consider: RIM's Blackberry OS[25], Apple's iPhone OS[25] and Google's Android OS[19].

2.1.1 Blackberry

RIM's Blackberry OS held 40% market share [25] at the end of 2009. It was known for its world class blackberry messenger service and corporate secure managed email. The

OS however, was not designed from the ground up with user application development in mind. A Java development environment was available but was reported to be extremely error-prone [18]. The software interface to low-level hardware features was extremely difficult and code did not port well among the large range of devices offered by RIM. Most concerning was our feeling that RIM engineers did not see their devices as a computing platform but simply as a communications platform. This feeling came after direct communications we had with engineers at RIM. We did not feel RIM's priority was on improving the application framework and for this reason did not continue with RIM's platform.

In the last three months of 2011 RIM's device sales account for only 6.5% of the smartphone market [24]. It is clear RIM has not kept up with the pace of innovation in the mobile space. As of Jan 1 2012 RIM stock has fallen 75% since the start of this work [11].

2.1.2 iPhone

Apple's iPhone was, and still is, the easiest of the three platforms for which to develop. It had 20% of the smartphone market share in early 2010 [25] and it seemed clear that it would only increase. iPhone apps are compiled directly into machine code which will naturally gave them a performance edge over the interpreted bytecode produced in the Java environment of Blackberry and Android. The key concern with the iPhone was wired connectivity to an external device. The dock connector is proprietary and was not accessible for user applications. Due to the high bandwidth nature of image and video content not having wired access to export data off devices and having no ability to add this functionality was of great concern at the time. This limitation was significant enough to forgo using this platform. It was subsequently decided to use a wireless interface to export data off the mobile device. As such, a wired interface should not have been a factor when selecting a mobile platform.

2.1.3 Android

In the middle of 2010, the Google Android OS had only 3.9% market share [19]. Despite this there was a good reason to believe the open source nature of the OS and the power of Google backing it would make Android a primary mobile platform for research moving forward. It was built with applications development in mind, had a usable development environment, great documentation and was a project in which Google was investing heavily. By the end of 2011 Android had 47% market share [21]. There were more than 750,000 android activations per day all around the world in December 2011 [22]. In the developing world more than 350,000 android phones were activated in Kenya in just a few months [13]. A Google Android smartphone has become the first computer for many people across the world. For these reasons we chose to do our work on the Google Android OS.

2.2 Phone and FPGA Hardware Background

In this section, we will provide a few relevant details about the devices that were used to implement this work. We will start with the mobile phone that was chosen: the Google Nexus One. Then we will discuss some details about the Android OS that is running on the Nexus One. We will then present some information about FPGAs and the specific development board employed.

2.2.1 Mobile Device

The device used throughout this thesis is the Google Nexus One. The core application processing chip is Qualcomm's first generation snapdragon system on chip (SoC), the QSD8250. it contains a 65nm ARM Cortex A8 processor running at 1Ghz [1]. ARM is currently the standard instruction set for low-power processors, and, as of 2011, all current Android devices run on ARM-based processors and 90% of all embedded smartphone

processors are ARM [12].

2.2.2 Android Operating System

The primary application development language in Android is Java [9]. Applications are written in Java, and then compiled into bytecodes[3], which are intermediate instructions, interpreted at runtime by a virtual machine[3]. Google has optimized a virtual machine and bytecode specification created by Dan Bornstein, known as Dalvik [3]. Its principal difference from a standard Java virtual machine is that Dalvik is a register-based architecture while Java is a stack-based architecture.

To help ease some of the performance overhead of interpreted bytecode, Google implemented a Just in Time (JIT) compiler for Android which was released with Android version 2.2 [32]. A JIT is a runtime optimization in which the virtual machine can algorithmically decide that a portion of bytecode should be fully compiled into machine code and run natively [28].

Google released the first iteration of the Native Development Kit (NDK) In June 2009 [5], which allowed developers to develop performance critical code in C/C++ and compile it directly into object code. The Java Native Interface (JNI) framework is used to interact with the compiled object code from a Java Android application [20]. Using the NDK, much of the performance overhead incurred through the Java abstraction could be mitigated at the cost of increased development time. Even with the JIT and common runtime optimizations enabled by interpreted code, our own measurements showed a 30x performance overhead compared to object code created with the NDK.

We measured the performance advantage of using the NDK to compile native object code by performing Gaussian convolution, an filter that blurs images. We coded this filter on a mobile phone in Java, with and without the JIT, as well as with native object code with and without parallel vector instructions. Table 2.1 shows the relative performance of the mobile device for all these cases. The huge performance jump between having the

JIT enabled and compiling to native code means that for any fair comparison between and Android mobile device and an FPGA the mobile device code must written with the NDK.

Mobile Gaussian Convolution Performance				
	Java	Java with JIT	NDK	NDK with Vector Instructions
Relative Speedup	1	1.5	41	64
Execution Time (s)	45	30	1.1	0.7

Table 2.1: NDK Performance Advantage on an Android Mobile Device

2.2.3 Field-Programmable Gate Arrays

FPGAs have widely been used in commercial digital systems for telecommunications infrastructure and many other applications. Their role as viable computational accelerators has remained, until recently, largely academic. Long and difficult development cycles made their use impractical when, for the better part of two decades, it was comparatively easy to design software that would naturally increase in performance with each new generation of CPU. In recent years, this steady increase of single threaded performance has run up against physical and technical limitations, which lead to the rise of multi-core CPUs. Developing for multi-core CPUshowever has proven significantly more difficult than for single core. As a result we are inclined to re-evaluate the computational mediums that are available other then traditional CPUs and find ways to utilize heterogeneous computing platforms in an efficient manner. It is possible that one component of the heterogeneous computing platforms of the future is an FPGA. There are already large-scale research efforts at Microsoft [26] and IBM [2] directed at compilation to heterogeneous systems that include FPGAs.

Modern FPGAs have a number of resources types that are accessible to the hardware designer. The most relevant to this work are Look Up Tables (LUTs) that implement boolean logic, hard multipliers and block memory. Each FPGA will have a fixed number

of these resources available across the chip as can be seen Figure 2.1 which is simplified view of a modern FPGA.

LUTs are configured to implement the logic functions of the user's design. Although multiplication circuitry can be implemented using LUTs, the demand for multiplication is great enough and the area efficiency advantage of custom multiplication circuity is great enough that they are included as a separate hardware resource. Effective use of an FPGA's resources is critical for computational accelerators as hard memories and multipliers are significantly more area efficiant they thier soft logic equivilents.

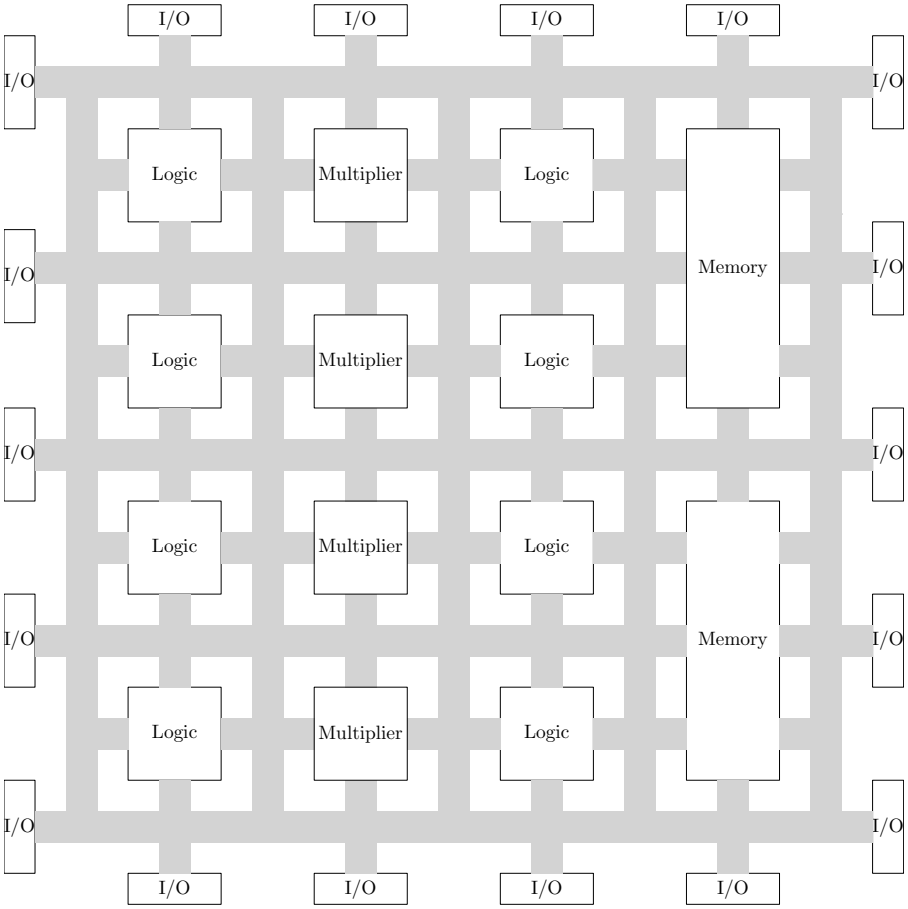


Figure 2.1: Heterogeneous FPGA

2.3 Computer Vision

Computer Vision is the science and technology of machines that see; that is, machines that can extract information about the world and their surroundings from a digital image or sequences of digital images. The field of Computer Vision is motivated by the common tasks that a human can do easily and yet computers struggle to do: Human drivers can identify and avoid pedestrians, obey traffic signs and navigate dense urban areas primarily via sight. Physiotherapists can assess joint and muscle injuries by looking at a patient's stability during certain exercises. People can readily detect never before seen objects as having a known purpose, such as being a chair or a table, despite a near limitless variety of colours shapes and sizes as the common characteristics are understood. These are the kinds of tasks that researchers in the field of Computer Vision hope machines will be able to perform as well or better than humans.

The field of Computer Vision has evolved into several sub-domains, which are characterized by common types of capabilities that researchers would like computers to be able to do. Scene Reconstruction is the ability to extract a three dimensional model of a physical space given a set of images of that space. Segmentation is the ability to partition an image into a set of related pixels. One common application of segmentation is to find the boundaries of physical objects with an image. Machine Learning is the ability for an algorithm to update its understanding of a problem when it is explicitly told that it was correct or incorrect in the past. Video Tracking is the ability to follow an object efficiently though time as it changes its position in the input images. Object Detection is the ability to determine whether or not a specific object is contained in an input image. Generally, Object Detection focuses on classes of objects while Object Recognition distinguishes between specific instances of objects in these classes. For example Object Detection might be used to find a face in an image; Object Recognition would then determine whose face it is specifically.

If these tasks can be performed by computers, then the output will become the input

data for more complex real world applications. The code complexity of these tasks is 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 a the open source Computer Vision project, OpenCV, 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. As the results in this work are based on the OpenCV project, we will describe it in more detail in the next section.

2.4 OpenCV

The Intel Open Computer Vision Project was launched in 2006 as collaboration between computer vision researchers and Intel [15]. It currently contains more than 2500 algorithms used in Computer Vision which are submitted by the open-source community and optimized by Intel to take advantage of low level capabilities of its CPUs. It is being used all over the world and has been downloaded more than 2.5 million times [15].

The code base is split into several modules; many of the sub-domains of Computer Vision discussed earlier are represented along with system-level functions and algorithms common to many sub-domains. Among the highest level and most directly useful functions provided are the ability to learn what an object looks like and then to look for that object in an image. The information that describes what an object looks like can be sent to the object detector along with new image and OpenCV can determent if the object exists in the image. If an object is detected OpenCV will return where it was located and what size it is. The focus of this work will be on object detection, a data intensive and computationally intensive function of OpenCV. We will accelerate this algorithm, using an FPGA for use on a mobile device. The generality and parametrization of the OpenCV implementation will be maintained in order to keep it as useful as possible to

many users.

2.5 Multi-Scale Haar Cascade Detection

The Multi-Scale Haar Cascade Detection (MSHCD) is the object detection algorithm used in OpenCV. It was introduced by Viola-Jones in 2001 [30]. The work was impressive for providing both high quality and relatively fast object detection in a fairly simple way. Since its publication, there have been more complex object detection algorithms proposed [14][23], but the method proposed by Viola-Jones still remains an active area of research due to its computational simplicity and high parallelism. There have been many works published exploiting different levels of parallelism inherent in the algorithm on various computation platforms, including CPUs [27] GPUs [10], custom hardware [6] and FPGAs [8][31]. Vision researchers attempt to make detection algorithms smarter by making more complex and intelligent decisions. At the same time, engineering research has been updating and evolving more simplistic approaches in order to achieve the greatest speed on available hardware platforms. The present work is an example of the latter but before we can discuss how it differs from other recent works we must must describe in detail the MSHCD algorithm.

The MSHCD algorithm take as input an 8-bit image and a classifier cascade which is a characterization of the object for which the algorithm is searching. When the algorithm has finished, the result is a list containing the (x, y) locations of each object it found, and a scale value which represents how large or small the object was in the input image. MSHCD, like many in graphics and computer vision, is an example of a sliding window application. Here calculations are done on a group of pixels in a small bounding box around a central pixel. This bounding box is known as a window. The center of this window moves to every location across the input as illustrated in Figure 2.2. At each location a calculation is done and the algorithm can access any pixel data that is contained

within the window. Typically the window will step in horizontal or vertical direction by one pixel.

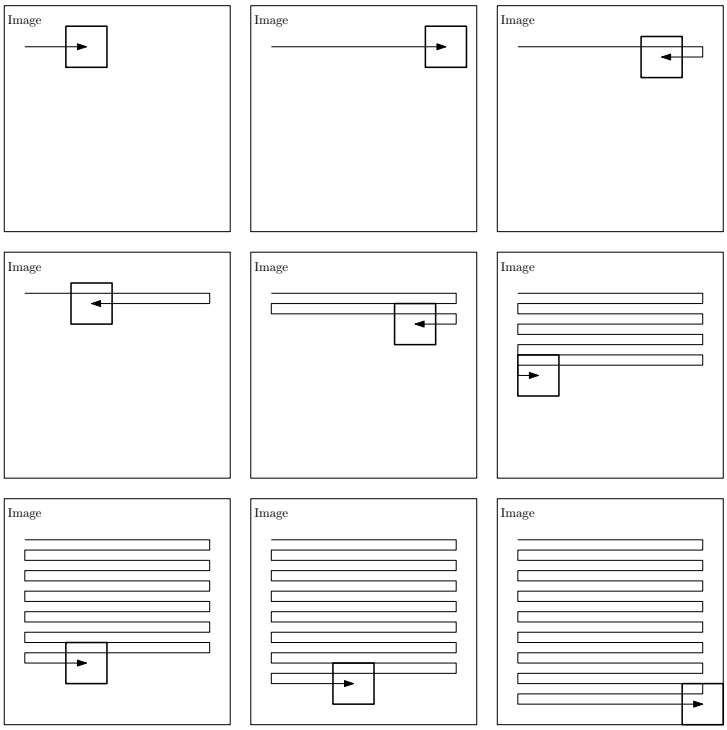


Figure 2.2: Window Sliding Across an Image

The internal computation of MSHCD consists of a number of steps which will be described below starting with the first step: scaling the image.

2.5.1 Multi-Scale

When looking for objects, a robust detection algorithm must account for variation in the size of the objects it is detecting. Objects closer to the camera will appear larger than objects further from the camera. Rather than learning what an object looks like at multiple sizes, or algorithmically scaling the model that describes how and object looks, MSHCD iteratively reduces the size of the input images. By doing so, objects bigger than the window will eventually become the size of the window if scaled with enough iterations. The scaling process can be seen in Figure 2.3; notice that the window size remains a fixed size. There is a tradeoff between the number of scales, the total

computational effort, and the quality of the results. Having more scaling iterations, and thus a smaller scaling factor will increase the quality of results at the cost of additional runtime as each scale is a new image that needs to be processed.

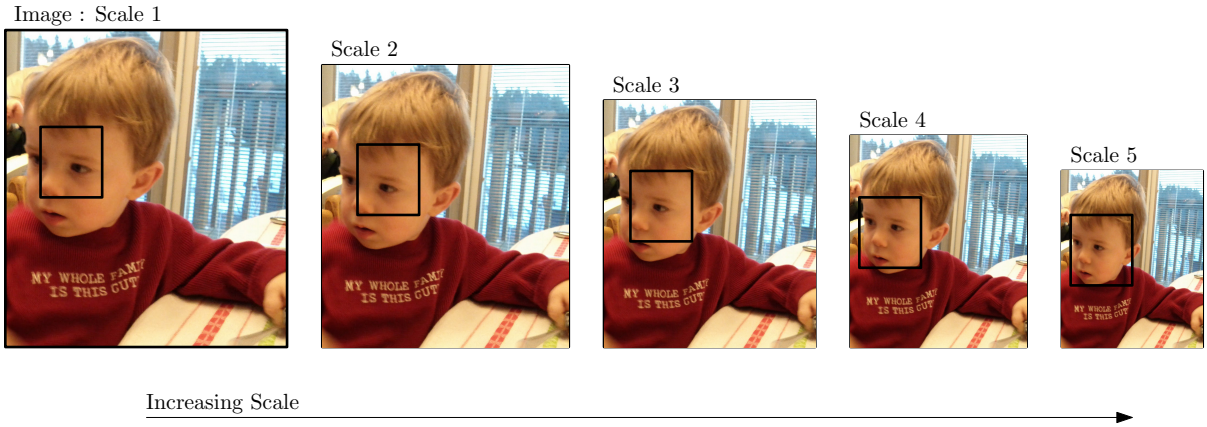


Figure 2.3: Image Scaling

To eliminate cumulative rounding errors, the MSHCD detection always starts with the original image when scaling, rather than scaling again a previously re-sized version of the image.

For example if the image is being scaled with a $ScaleFactor$ of 1.1 (10%) with each iteration of the algorithm than the third iteration would scale the original by $1.1^3 = 1.33$. It rounds the scaled image dimensions and scaled pixel values to whole numbers. When it starts the next scale iteration, these rounding errors are not carried through as the process will start again with the original image. The exact operations required to perform image scaling just as MSHCD are outlined below.

First the $TrueScale$ is calculated, this is the exact amount that the algorithm would ideally scale the image which is based on the loop iteration i and the $ScaleFactor$

$$TrueScale = ScaleFactor^i$$

Next the height and width dimensions of the scaled image are calculated, which are rounded to the nearest whole number.

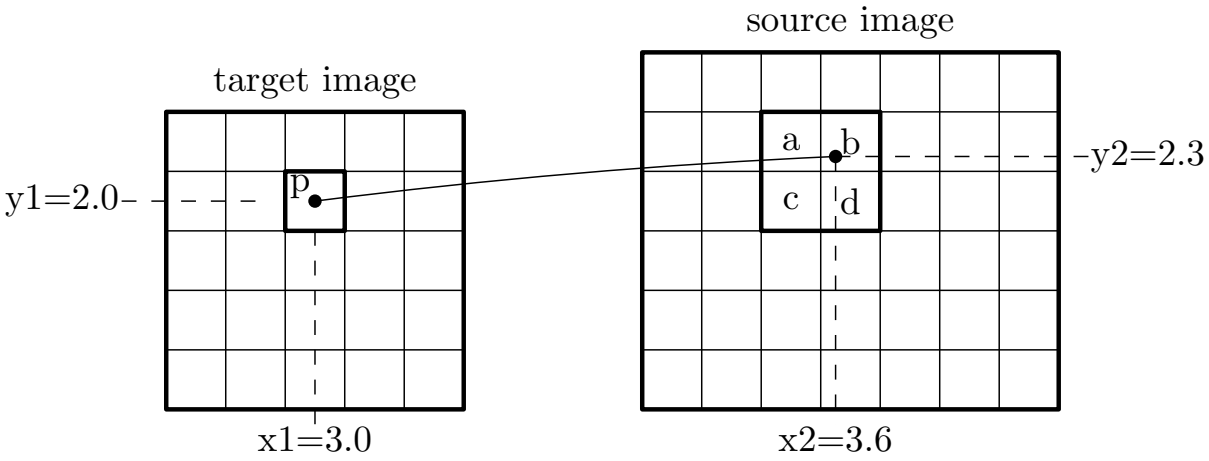
$$smallImageWidth = Round(fullImageWidth * TrueScale)$$

$$smallImageHeight = Round(fullImageHeight * TrueScale)$$

Once the final scaled image dimensions are known, new scale factors are calculated based on the actual size difference between the original and scaled image including the rounding errors.

$$scaleX = Round(smallImageWidth/fullImageWidth)$$
$$scaleY = Round(smallImageHeight/fullImageHeight)$$

To calculate the new pixel values in the scaled image, linear interpolation is used. Every pixel in the scaled image is calculated from a weighted average of 4 pixels in the original image. Figure 2.4 illustrates this computation. The scaled image needs the value of a pixel in the original image at location y=2.3 and x=3.6 but that, of course, is not a discrete location in the image. The scaled pixel value is computed as a function of the discrete pixel values nearby. If y is 2.3 the pixel value at y=2 would be weighted 70% and the value at y=3 would be weighted at 30%. The final value is then rounded to the nearest 8-bit number. The next step in the MSHCD algorithm is to transform the newly scaled image into an integral image, this is presented next.



$$p = (1 - \Delta y)[a * (1 - \Delta x) + b * (\Delta x)] + (\Delta y)[c * (1 - \Delta x) + d * (\Delta x)]$$
$$newPixel = Round(p)$$

Figure 2.4: Linear Interpolation Image Scaling

2.5.2 Integral and Square Integral Images

Transforming an image into an integral image makes it possible generate the sum of all the pixels in *any* rectangular region of the original image while only reading 4 values from the integral image. When later stages of the MSHCD detection algorithm are presented it will become clear how valuable this property is.

The integral image is computed such that every pixel (x, y) in the new image is the sum of all the pixels with indices less than (x, y).

$$IntegralImage(x, y) = \sum_{i=0}^{i<x} \sum_{j=0}^{j<y} Image(i, j)$$

An illustration of this can be seen in Figure 2.5. If we look at just the corners of a rectangle in the integral image we can find what the sum of the pixels contained in that rectangle would have been in the original image by simple arithmetic. This can be seen by inspection of Figure 2.6. As a result of this the sum of pixels in a given area of the images can now be performed in a constant time with 4 memory accesses.

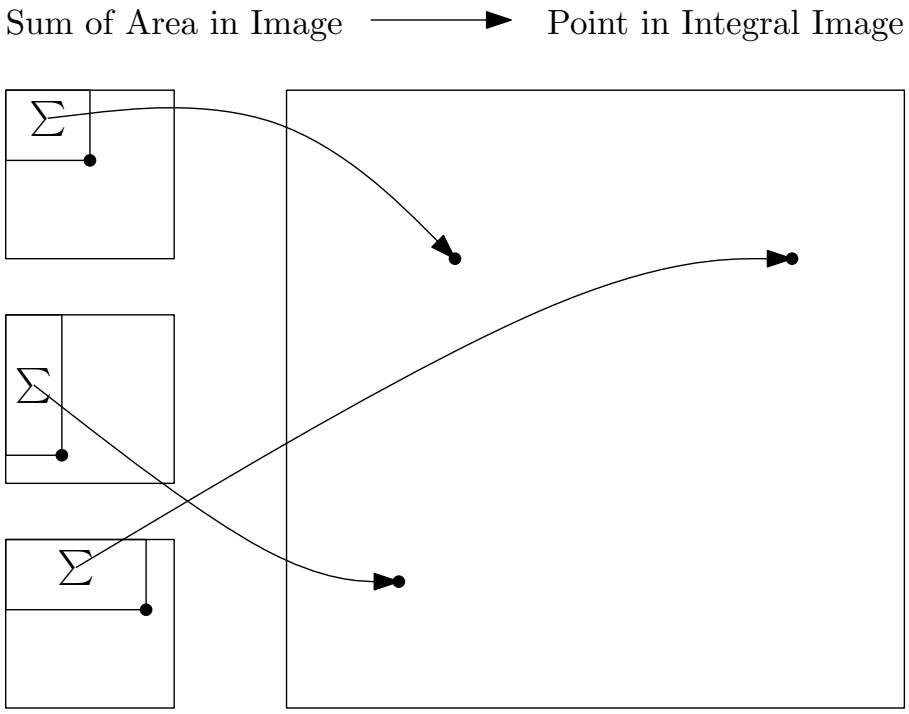


Figure 2.5: How to Construct an Integral Image

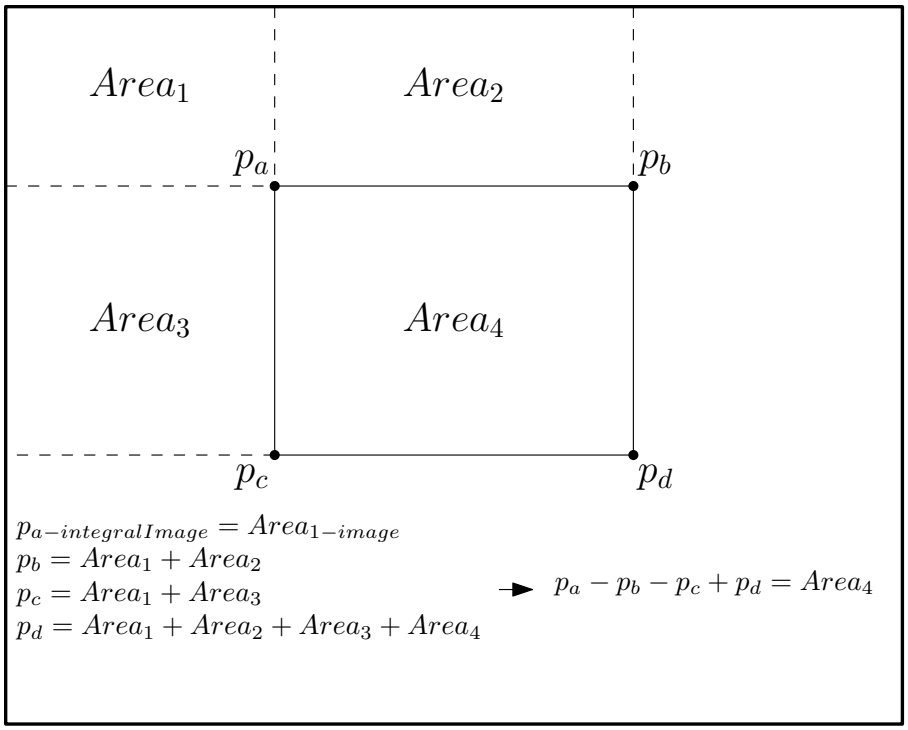


Figure 2.6: Principal Integral Image Property

The square integral image is identical to an integral image except every pixel value from the image is squared first. The square integral image lets you access the sum of the squares of the pixels in a rectangular region of the original image in 4 memory accesses. The square integral image needed for a computation that will be discussed later in this section.

2.5.3 Haar Features

Haar features sit at the core of the MSHCD algorithm. A Haar feature is a standardized way of describing a very simple property of a window. Multiple Haar features, in aggregate, can describe complex properties of a window, including whether or not that windows contains the object that we are trying to detect. There are many types of Haar features but OpenCV limits itself to using only the basic five as illustrated in Figure 2.7, and as shown each is constructed out of either two or three rectangles. The size and location of these rectangles are specified relative to a window as seen in Figure 2.8.

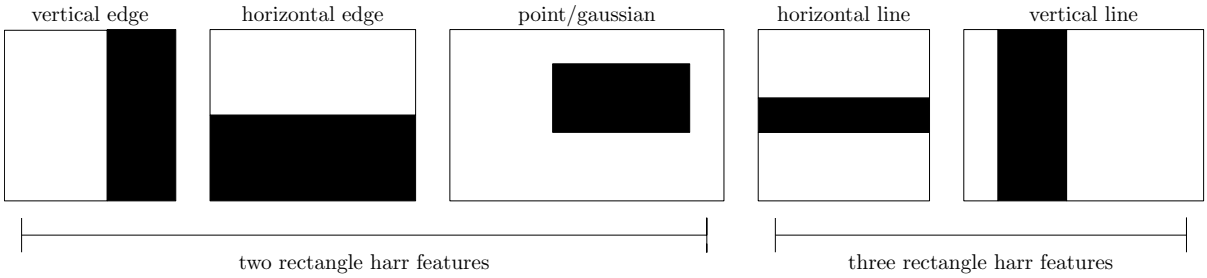


Figure 2.7: Five Supported Haar Feature Types

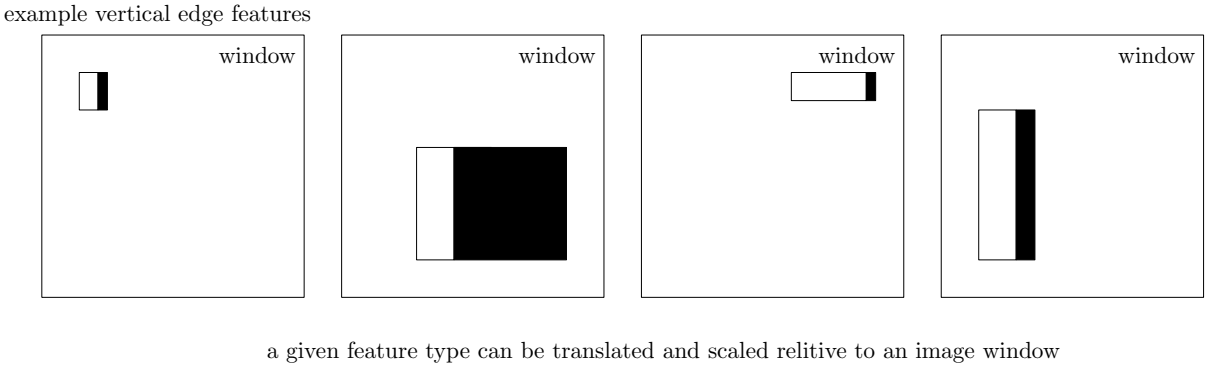


Figure 2.8: Variability among One Haar Feature Type

A specific Haar feature is characterized by describing rectangles with starting x and y locations, and a *height* and a *width*, as illustrated in Figure 2.9. As well as the variables seen in Figure 2.9, each rectangle in a Haar feature has an associated weight.

If one was looking for a face, there may be features such as those in Figure 2.10 looking for the eyes and mouth at those particular locations relative to a window. One might expect that pixels in the black rectangle covering the mouth in Figure 2.10 would, on average, have different values than those in the clear rectangle surrounding it. If in that location of that window this was true it would be more likely that the window is in fact a face. To test a feature, the sum of all the pixels in the first rectangle is multiplied by the feature rectangle's weight and added to the sum of all the pixel values in the second rectangle multiplied by its weight. The result is then checked to see if it is within the threshold learned previously during training. Recall from the previous section that the integral image allow the efficient computation of the sum of pixel values in a given rectangle. The test of every Haar feature requires the summation of a region of a window

and this is why the MSHCD algorithm first generates an integral image. A single feature on its own can not describe a very complicated object so to describe more complex ones multiple features are bundle together in packages called classifiers.

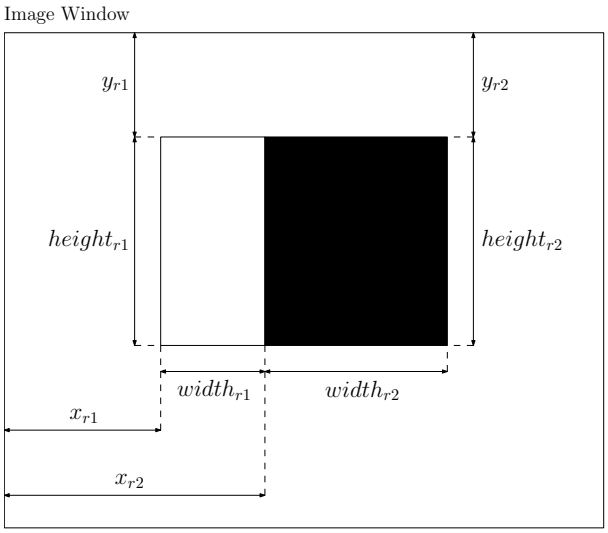
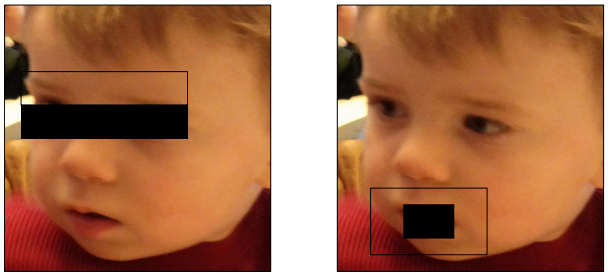


Figure 2.9: Feature Characterization



vertical edge feature filtering for eyes

point feature filtering for mouth

Figure 2.10: Application of Haar Features

2.5.4 Haar Classifiers

A Haar classifier is a collection of Haar features that together describe a more complex object. A window is tested with each feature in the classifier in the manner described in the previous section. Each feature has a value it returns when it passes, $alpha_1$, and a value it returns when it fails, $alpha_2$. The value returned by each feature is summed

and if the total for *all* features in the classifier is greater than the classifier threshold the classifier is said to have passed.

Figure 2.11 illustrates the hierarchy of values in a Haar classifier as well as the final structure that will be discussed in the next section which is a collection of Haar classifier known as a cascade.

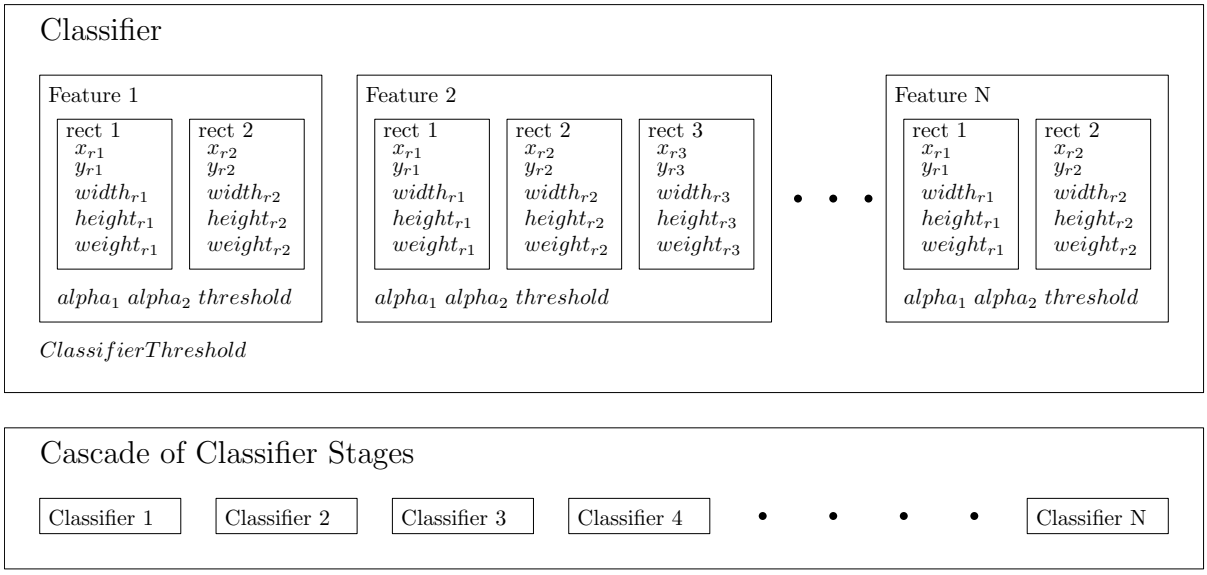


Figure 2.11: Data Structure of a Cascade of Classifier Stages

2.5.5 Cascade of Classifiers

In order to detect complicated objects with high confidence we would like classifiers with large numbers of features. As the number of features increase in any given classifier so does the computation required to evaluate it as every feature must be evaluated. Rather than have one very large classifier, the MSHCD uses a cascade of many varying sized classifiers. The classifiers start with a small number of features and gradually increase in size. As an example, Figure 2.13 shows the average number of features in each classifier stage for all standard OpenCV classifier cascades. If a window passes the first classifier

stage, it is sent to the second and so forth. If any classifier stage fails, the object is deemed not to have been detected in the window and the process starts again on the next window. This is illustrated in Figure 2.12. This mechanism is efficient because it allows the algorithm to reject a window quickly after the earlier, smaller, classifiers if the window looks nothing like the object that is being detected. This represents a significant performance advantage when in many cases one can assume the vast majority of windows in an image will not contain the object for which the search is being conducted.

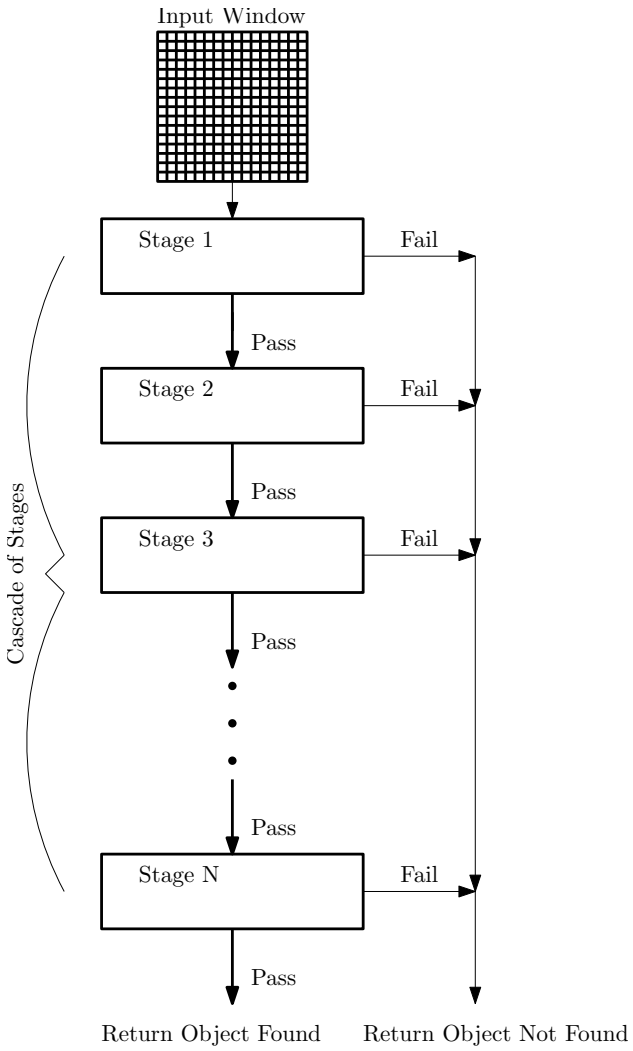


Figure 2.12: Computation of a Cascade of Classifier Stages

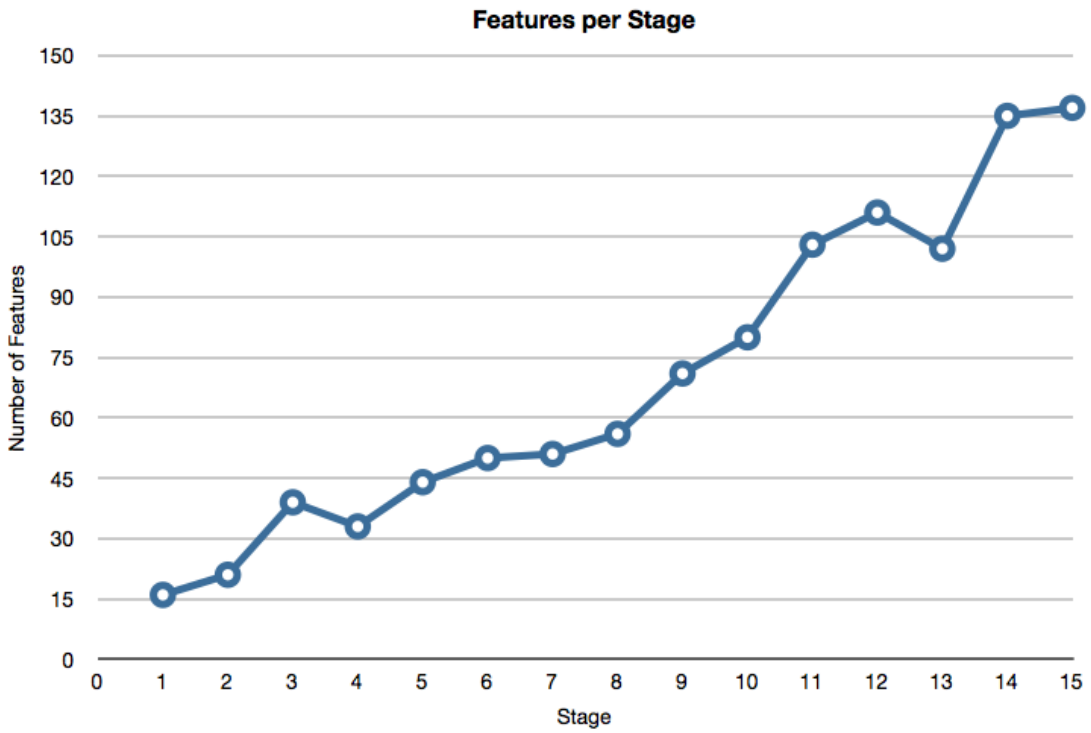


Figure 2.13: Average OpenCV Feature Distribution in a Cascade

2.5.6 Variance Normalization Factor

A picture of the same object that is underexposed or was taken in low light will, on average, have lower pixel values than the same picture taken in good lighting or overexposed. However, when Haar features are being evaluated sums, of window pixels are being compared with a previously learned threshold value. In order to compensate for the pixel value differences due to lighting conditions, the MSHCD computes a value known as the variance normalization threshold. This value normalizes threshold values for the lighting conditions of the specific input window. It is calculated for every new input window.

$$VarianceNorm = \sqrt{(\sum_{i=0}^{i<x}\sum_{j=0}^{j<y} window(i,j))^2 - (\sum_{i=0}^{i<x}\sum_{j=0}^{j<y} window(i,j)^2))}$$

The sum of all the pixels in the window can be quickly calculated with the integral image and the sum of all the squared pixels of a window can be calculated using the

square integral image. The calculation of the variance normalization factor is the reason why a square integral image is generated.

2.5.7 Computational Complexity

The computational complexity of the MSHCD algorithm is a function of the image size and the total size of the cascade of classifiers. By using an integral image every feature can be applied in a constant amount of time. By using a square integral image the variance normalization factor can also be computed in constant time. Due to the use of these image transforms the amount of computation is decoupled from the size of the image window. The number of total windows that need to be processed is roughly equal to the number of pixels in the image M x N. Many of these windows will fail at a very early stage of the cascade and a few will fail at a very late stage of the cascade. Therefore on average each window will require F features from the cascade, where the minimum value of F is the number of features in the first classifier and the maximum value is the total number of features in the entire cascade. The total computational complexity is then O(F x M x N).

2.6 Previous Hardware Implementations

There have been many attempts to implement some variation of the Haar detection algorithm in hardware, both in hard silicon and on an FPGA. In these works, there are system and platform level optimizations to increase performance or reduce resource utilization under a variety of operating conditions but principally they all contain, and are performance limited by, some variant of the structure illustrated in Figure 2.14. The performance of this algorithm is rate limited by effective memory bandwidth accessing two data elements: the location of the classifier coordinates, (x_r, y_r) in Figure 2.11 and integral image. Figure 2.14 is a representation of the most basic structure with a single

processing element (PE). In this structure it can be guaranteed that in each cycle a new (x_r, y_r) can be read from the feature memory and subsequently used as an index into the integral image during the next cycle in a pipelined fashion. Previous works, architecturally, have evolved this basic structure in order to exploit the parallelism inherent in this algorithm. These variants will be discussed in the following sections.

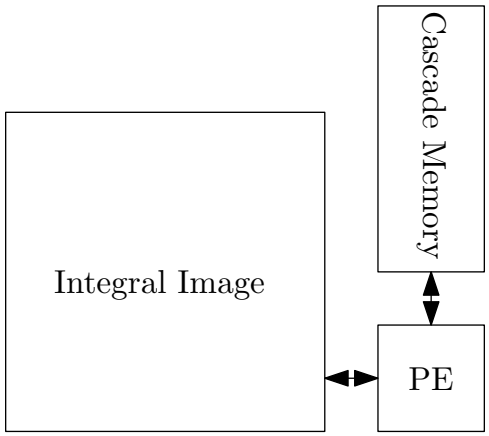


Figure 2.14: Basic Compute Architecture

2.6.1 Cascade Replication With Window Caches

For all but the smallest input image sizes, having multiple copies of the integral image on-chip is simply untenable. However it is reasonable to have many smaller memories large enough to store only single window; these memories are called window caches and are associated with a PE. Also, depending on how many features are in the total classifier cascade, it can be a relatively small memory and it would makes sense to have multiple copies of it on the device. This would allow multiple window caches to be processed independently and in parallel as each window cache could independently access one of the copies of the classifier cascade. This architecture is illustrated in Figure 2.15. Control circuitry loads a window cache with a copy of data from the integral image and instructs the processing element to begin. It then starts loading the next window cache and instructs the second processing element to begin. This method is a performance

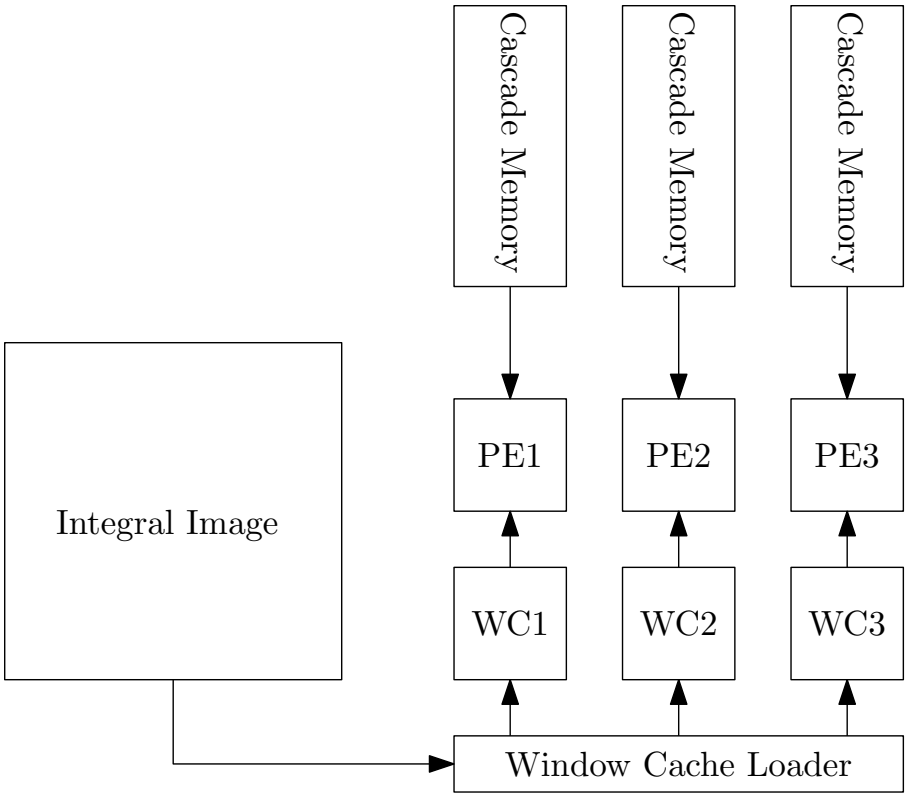


Figure 2.15: Feature Replication Compute Architecture

improvement over previous versions, but it fails to optimize for the common case: when windows fails at a very early stage. A typical window size is 20x20 pixels; this means to fill a window cache would require 400 reads from the integral image. Thus in this architecture if a PE fails in fewer than 400 cycles, the subsequent PE's window cache will not yet be loaded. The two PE's would not be doing useful computation in parallel. On the other hand if a PE takes longer than 400 cycles to fail, the subsequent PEs windows cache will have had time to load and both PEs will be working in parallel.

2.6.2 Replicated Windows with Load Balancing

This implementation [7] split up the cascade memory into sections, increasing the number of read ports into the cascade without increasing the actual number of memory bits. Each PE had access to its own window cache which stores *identical* data as all other window caches. All the window caches are loaded simultaneously with *identical* data

and each PE applies a different part of the cascade to the *same* window. Each section of the cascade could be accessed by two PEs as illustrated in Figure 2.16. In an effort

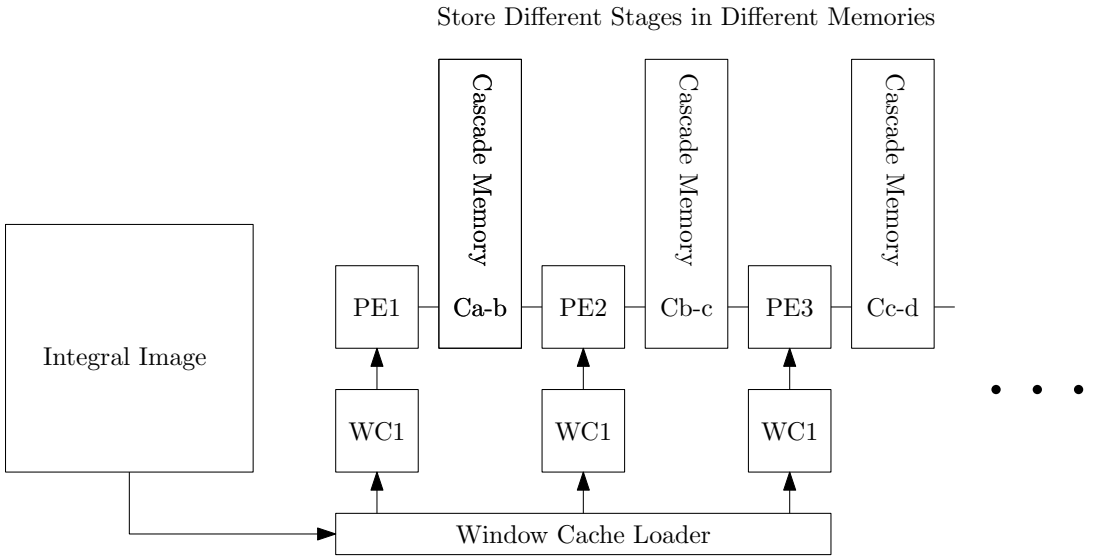


Figure 2.16: Memory Based Replicated Windows with Load Balancing

to avoid idle processing elements the system kept track of when and what PEs were idle. Load balancing was then used at run time to try and keep the most PEs possible simultaneously busy and thus increase performance. This implementation doesnt address how long it takes to load new data into the window cache and is ultimately limited by the performance of that processes.

2.6.3 Register Based Windows and Partial Cascade

To dramatically increase how quickly the processing hardware could access data, a method was proposed that stored the window and some of the cascade in *registers* rather than memory [16]. In this method, a parametrized number of classifiers are evaluated on a single window completely in parallel. Since 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 virtually instantly. If it did, not this method would fall back to reading a cascade

memory and computing the remaining classifiers sequentially. The features executed in parallel are initialized into registers at the time the circuit is synthesised. For these reason if this hardware implementation changed what object it was looking for, and thus what cascade it was using, the entire system would need to be resynthesised. Like the *Replicated Windows with Load Balancing* method described above, this method utilizes additional hardware to compute a single window as fast as possible. This architecture is illustrated in Figure 2.17. There are many parallel feature evaluators, and one generic

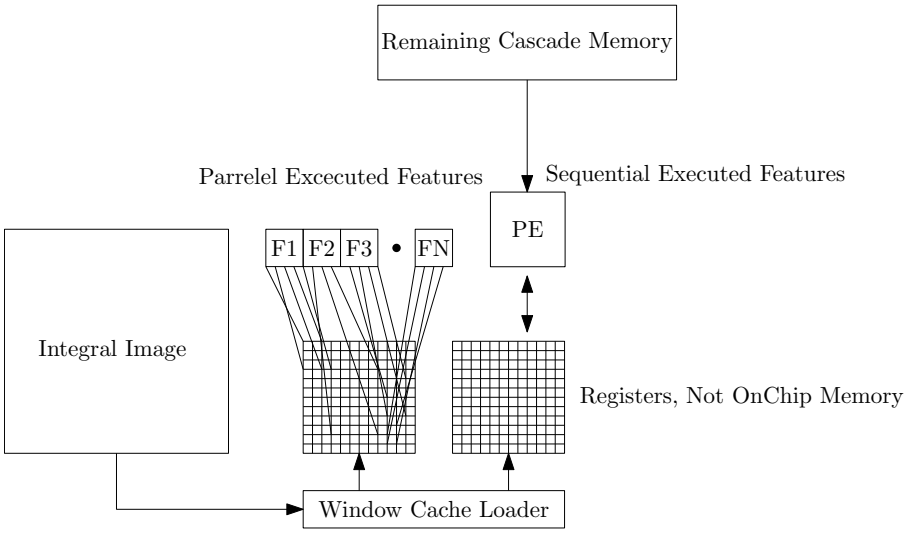


Figure 2.17: Register Based Windows

sequential processing element. This implementation performs extremely well when there is high register and logic usage, meaning many features are computed in the parallel section. It slows down quickly with reduced resource utilization. This work is the most complete system that focused on large image sizes, large classifier cascade sizes and appropriate scaling factors while not compromising on quality of results found in the prior work to date. By loading new data intelligently in parallel and processing that data in parallel this implementation successfully addresses the two bottlenecks of computation and achieve significant performance gains as a result. This system was able to achieve, on average, processing one window every 5 cycles which is fast enough to be comparable to our work. Section 4.6 will give a detailed performance comparison between this work

and our work normalizing the results as much as possible.

2.7 Summary

In this Chapter we began with an overview of the available mobile platform choices, mobile device hardware and FPGA technology. The field of Computer Vision was then introduced alongside a powerful open source software library that implements many key algorithms in the field. The OpenCV MSHCD algorithm was then presented in detail. Finally, a few key FPGA implementations of similar object detection algorithms were presented. In contrast with previous work on FPGAs that accelerate the processing time for a single window, this work will focus on having a greater number of windows processing in parallel. The architecture for our FPGA implementation will be the focus of the next Chapter.

Chapter 3

OpenCV Compliant Multi-Scale Haar Cascade Detection

This Chapter will present the main contribution of this work which is a parametrized FPGA-based system for performing OpenCV compliant MSHCD. Recall from Chapter 1 that our definition of OpenCV compliant is, given the same input stimuli, the FPGA implementation should produce the same results. Our system uses the exact cascade classifiers format produced by OpenCV and when given an 8-bit input image produces exactly the same results as the OpenCV software implementation.

This Chapter will begin with a full overview of the hardware system. It will then discuss the two key innovations of this work. The first, was a new single instruction multiple data (SIMD) architecture which allows many windows to be processed in parallel with only one copy of the cascade memory on-chip. The second is an efficient way to load multiple window caches in parallel that made more intelligent use of every read from the integral image. Finally, it will outline the considerations that were made to manage fixed point arithmetic error in order to achieve OpenCV compliance.

3.1 System Overview

In this Section, we outline the high level structure of the MSHCD hardware design to set an overall context. Subsequently we will describe the architecture of the core computation that performs the MSHCD algorithm. There are many specific components involved in making the system OpenCV compliant that are neither performance-critical nor particularly novel. These components will be presented but not discussed in great detail. It is worth noting that many previous works do not build a complete vision system, but rather focus on the inner kernels only. In this work we have built the entire system that takes as input a single image and outputs the locations and sizes of all detected objects. Once we have described the complete system in Section 3.1.1, we will focus on the MSHCD in Section 3.1.2.

3.1.1 Top Level System

An overview of the data sent to and from the FPGA is given in Figure 3.1. The main storage elements needed by our implementation of MSHCD are described below. First a software function running on a host computer sends a cascade of classifiers to the device via a USB connection. These define for which object the hardware will search. The input to this software function is a stock OpenCV classifier cascade identical to those that are provided with the OpenCV source code. Next, another software function sends an 8-bit image to the system over the USB connection. The hardware system then processes the image and writes results, positions and scales of detected objects, to a FIFO which is transmitted back upon request to the host processor via USB.

Image Memory

The input 8-bit image is stored in the on-chip memory of the FPGA. As in OpenCV, it is stored in an 8-bit grey-scale representation. This means for each pixel of the image

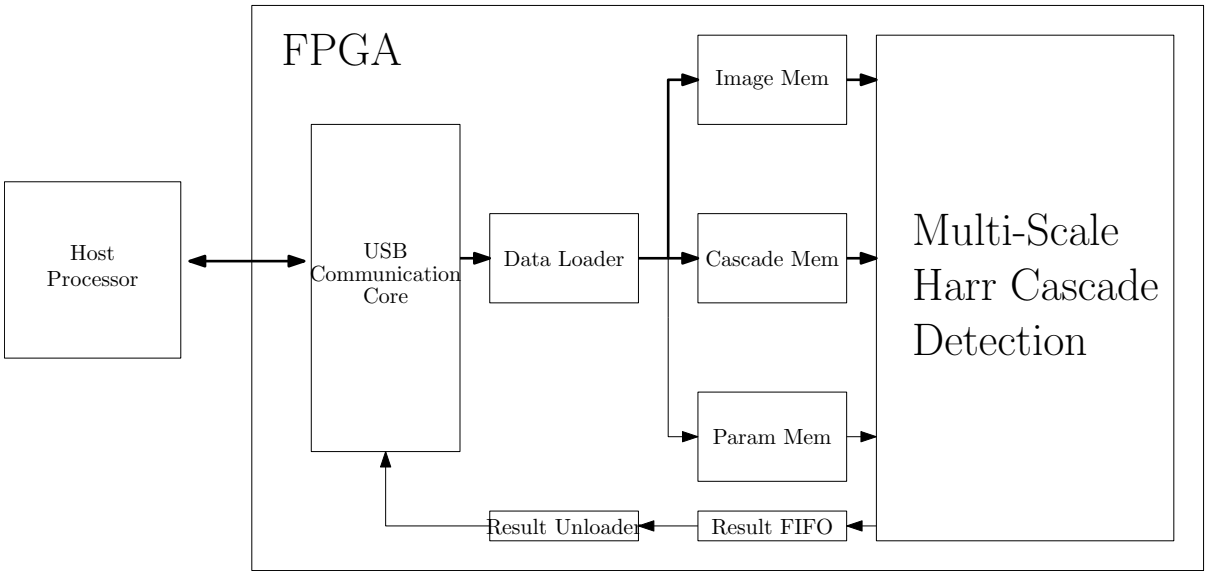


Figure 3.1: Top Level Hardware Diagram

only a single 8-bit value between 0-255 is required and represents the brightness of the image at that pixel. After the computation on one image is complete, a new image (the next frame) is loaded.

Both the maximum size of the input image *width* and *height* in pixels as well as the number of bits per pixel are hardware parameters. The maximum value of these parameters is constrained by the relevant capacity of the target FPGA.

Cascade Memory

In Chapter 2 we discussed the internal structure of a cascade of feature-based classifiers used in the MSHCD algorithm, and this was shown in Figure 2.11. For our hardware implementation the *cascade memory* is split internally into two separate memories: *classifier memory* and *feature memory*.

The *classifier memory* stores all of the non-integer values shown in Figure 2.11, which is all the *alpha* values, feature *weights* and threshold values. Interleaved between these values are control values that indicate to the hardware how many features are in the current classifier stage. Almost every value in this memory will be a fixed-point repre-

sentation of a floating-point number. Note from Figure 3.2 that, on average, only 5 reads from *classifier memory* are needed for each feature.

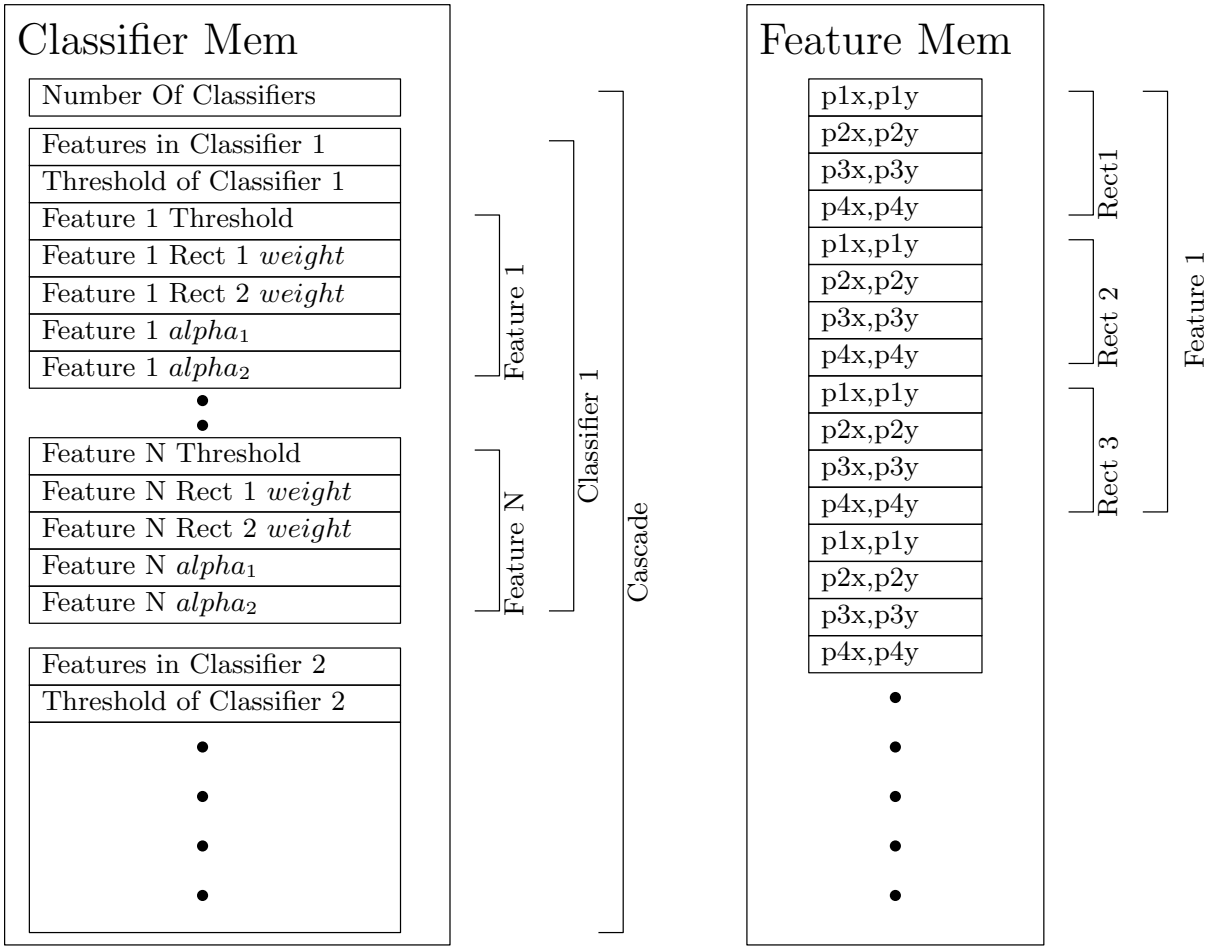


Figure 3.2: Feature and Cascade Memory Structure

The *feature memory* stores the (x, y) positions of the corners of each feature rectangle. As discussed in Chapter 2, the basic building block of the MSHCD algorithm is a Haar feature, which is a set of rectangular regions positioned relative to an image window. When using integral images, only the 4 corners of each rectangle needs to be fetched from the window of the integral image. So, in effect, the values in the *feature memory* act as an index into a window. In Figure 3.2 we can see that for each feature rectangle we will have 4 coordinates to access in a given window.

The cascade memory is split in this way so that in every cycle one coordinate can

be read from the feature memory and subsequently that coordinate can be used to read one value from a window. By decoupling the control information, weights and thresholds from the feature rectangle locations we can guarantee that during operation the system is reading one value every cycle from the window which is the maximum possible rate in our system.

The *feature memory* and *classifier memory* consume a significant fraction of a modern FPGA's on-chip memory for real world cascades which can contain thousands of features, requiring hundreds of kilobytes to megabytes of memory. Due to their size it is impractical to have more than one copy available on chip. Both memories are sized to guarantee they are large enough to store a cascade of N total features, where N is a compile-time system parameter of the hardware system.

Parameter Memory

At the top level of the system is a very small memory which stores all of the software defined parameters that are configurable at run-time. A list and short description of each can be seen below.

1. Input Image Width : Specifies the width of the image, in pixels, to be processed. This value must be smaller than the maximum image width (a compile-time parameter) supported by the Image Memory.

2. Input Image Height : Specifies the height of the image, in pixels, to be processed. This value must be smaller than the maximum image height (a compile-time parameter) supported by the Image Memory.

3. Scaling Factor : Specifies the percentage the image should be re-sized between each scale iteration discussed in Section 2.5.1 and shown in Figure 2.3. Decreasing the scaling factor will result in higher quality of results at the cost of a significant increase run-time.

4. Stages : Specifies the number of classifier stages of the cascade which must pass to result in a successful detection. If set to 0, the system requires that all of the stages of the loaded cascade must pass. This value gives the user an ability to quickly adjust the quality of the detection results. By lowering the number of stages that must pass the algorithm, it is more likely to detect each instance of the object that it is searching for in the image, at the cost of increase false positives. Depending on the demands of the application this can be worthwhile.

Output Result FIFO

The final result of the MSHCD hardware is a list of (x, y) coordinates of detected object positions with an associated scale. The scale value indicates what size was the image when an object was found at that (x, y) location. These results are exactly equivalent to an OpenCV data structure just before a final function is run called groupEPS(). This function scales all coordinates to their position in the full size image and then groups together all the clustered positive results. There is in many cases several positive results all in very close proximity of each other when an object is found in an image, and this function recognizes that these results represent a single physical object and returns one (x, y) in place of many for each object. In our system groupEPS() is run in software on the output of the hardware results.

3.1.2 Main Computation Hardware

This Section describes the flow of the main MSHCD hardware computation and links the hardware back to the algorithm presented in Chapter 2. We will then describe the details of the high level components of the main computation which are shown Figure 3.3.

The components in Figure 3.3 follow the procedure outlined in Chapter 2. The input image is successively scaled to smaller sizes and then an integral and square integral image is generated from the scaled image. A set of windows, offset by 1 pixel horizontally, from

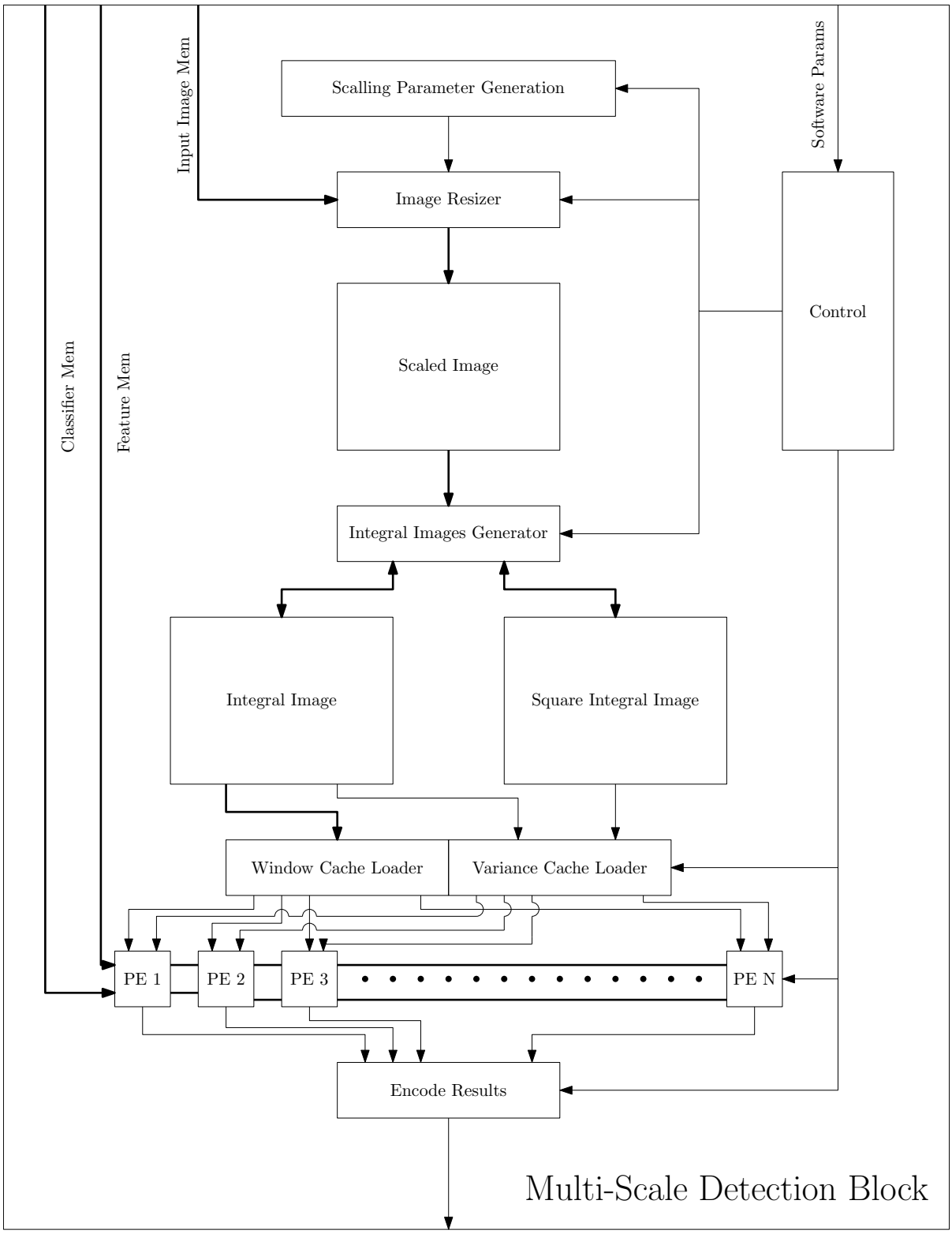


Figure 3.3: Top Level of Computation Hardware

the integral image are then loaded into small memories inside each processing element (PE). Then the same Haar features from the cascade of classifiers are applied to each window in each PE *simultaneously*. If any of the current windows successfully pass all of the classifiers in the cascade then an object was detected in that window and that information in encoded and sent out to the result queue.

Scaling Parameter Generation

This block calculates all the values needed to re-size the input image. These value were explicitly outlined Section 2.5.1. This hardware block reuses a single divider and multiplier to generate these values with minimal resources utilization, as it is not performance critical.

Image Resizer

This block preforms a linear interpolation style of image scaling whereby every pixel in the target image is calculated from a weighted average of 4 pixels in the source image as was described in Section 2.5.1. The hardware block time-multiplexes a single hard divider, hard multiplier and several adders to generate these values with minimal resource utilization. It computes the result at a rate of one output pixel every 4 cycles.

The process of scaling is not performance-critical. Once the integral image is generated from a particular scaled image, the scaled image for the next iteration is computed while the PEs are using windows from the current integral image.

Scaled Image Memory

A copy of the scaled image in stored in the on-chip memory of the FPGA and, as in OpenCV, it is stored in an 8-bit gray-scale representation.

Integral and Square Integral Image Memory

These two blocks of on chip-memory are the largest consumers of memory in the design. The number of bits used to represent each pixel is a function of the original image bit-width and the maximum supported image size. As described in Section 2.5.2, the integral image at any point is the sum of all the values of the scaled image above and to the left of that of that point. The highest value that the integral image will take will always be the pixel in the bottom right as it will be the sum of every pixel in the original image. This means the maximum value of the integral image would be the maximum value of a single pixel multiplied by the total number of pixels in the image.

In OpenCV, the bit-width of the integral image is set to 32 bits and the bit-width of the square integral image is set to 64 bits. Since we know the maximum size of the input image, the hardware can support and the maximum input image bit-width we make sure to only use the number of bits required. For a maximum image size of 320x240 with 8 bits per pixel, our system would generate an integral image block with a bit-width of 25 and a square integral image with a bit-width of 33.

Integral and Square Integral Image Generator

This block generates the integral image and the square integral image from the scaled image as was shown in Section 2.5.2. Although each pixel in these memories represents the sum of many pixels in the scaled image, the value can be computed at a rate of one pixel per cycle. This is the standard way to produce these images and it is algorithmically similar in OpenCV. Each target pixel of an integral image can be inferred from the target pixel in the scale image plus the running sum of the row of the scale image plus the pixel in the integral image one pixel above the target, which has previously been computed.

Window and Variance Cache Loader

The window cache loader efficiently loads window data from the integral image into window caches. How it does this will be discussed extensively in Section 3.2.1 of this Chapter as it is vital to overall system performance. Alongside it is a non-performance-critical but important block that loads the data required to calculate the variance normalization factor, discussed in Section 2.5.6, associated with each window. To calculate this, the pixel values of the corners of the current window are needed from both the integral and square integral image. This block loads those eight values into a very small memory inside each PE so that the variance normalization factor can be calculated while the window caches load completes.

Processing Elements

The processing element is the block that applies the classifier cascade to a specific window and determines how many classifiers in the cascade the window passes. The basic components can be seen in Figure 3.4. These include a double buffered window cache and a double buffered variance cache to permit overlapped loading and computation. This also includes the variance normalization factor calculation hardware and the feature evaluation hardware. The feature evaluator is guaranteed to operate at a rate of one lookup into a window cache per cycle and will stop at a classifier stage boundary if the current window failed that stage, as illustrated in Figure 2.12. The number of PEs in the system is a compile-time parameter called *Cores*.

3.2 Hardware Design Details

In this section we will described more of the significant details of the hardware that are critical to the overall performance of the system.

The evolution of hardware implementation of Haar detection has tended towards more

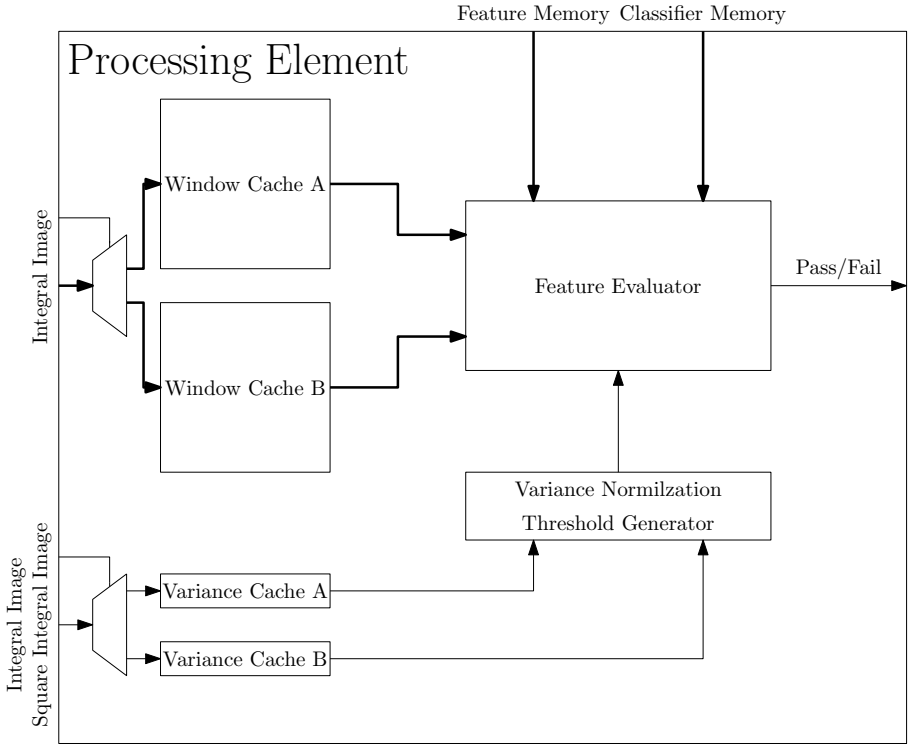


Figure 3.4: Integral Structure Of Processing Element

efficient use of on-chip memory, principally the more efficient use of an integral image and a cascade of classifiers. If this data could be replicated this algorithm can be massively accelerated due to its near perfect parallelism. The register window based method in [16] and discussed in Section 2.6.3 exploits this near perfect parallelism by storing windows and features in registers and running many feature evaluators on a single window in parallel. This method will result in many wasted calculations and needlessly use extra energy. Since it is computing features in parallel, this method is unable to stop processing when the window has failed a classifier, as the other classifiers are already computing at same time.

The present work exploits a different form of parallelism. We will present an architecture that makes very efficient use of each read from on-chip memory and only performs useful computations. Our architecture was developed from one simple observation: neighbouring windows look very similar, an illustration of which can be seen in Figure 3.5.

Figure 3.5: Commonality Between Neighbouring Windows

From this observation we can be certain that a single pixel from an image (or more accurately an integral 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:

1. Parallel Window Cache Loading; and

2. SIMD Processing.

3.2.1 Parallel Window Cache Loading

Ensuring PEs have data to work on can quickly become a bottleneck in the system and limit the useful number of PEs. To load a 20x20 window one pixel at time will take 400 cycles. By design, a large number of windows should fail very early on in the classifier cascade. For OpenCV cascades the first 4 to 5 classifiers can be read in roughly 400 cycles. 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 data.

In this work we make use of the common information that we notice in neighbouring windows to enable our vastly superior average time per loaded window cache. Figure 3.6 illustrates more concretely how we can take a single read from the integral image and fit it into multiple window caches. In this example we notice that the first value in the first row of window one is 1 and the final value of the first row of window three is 8. This means

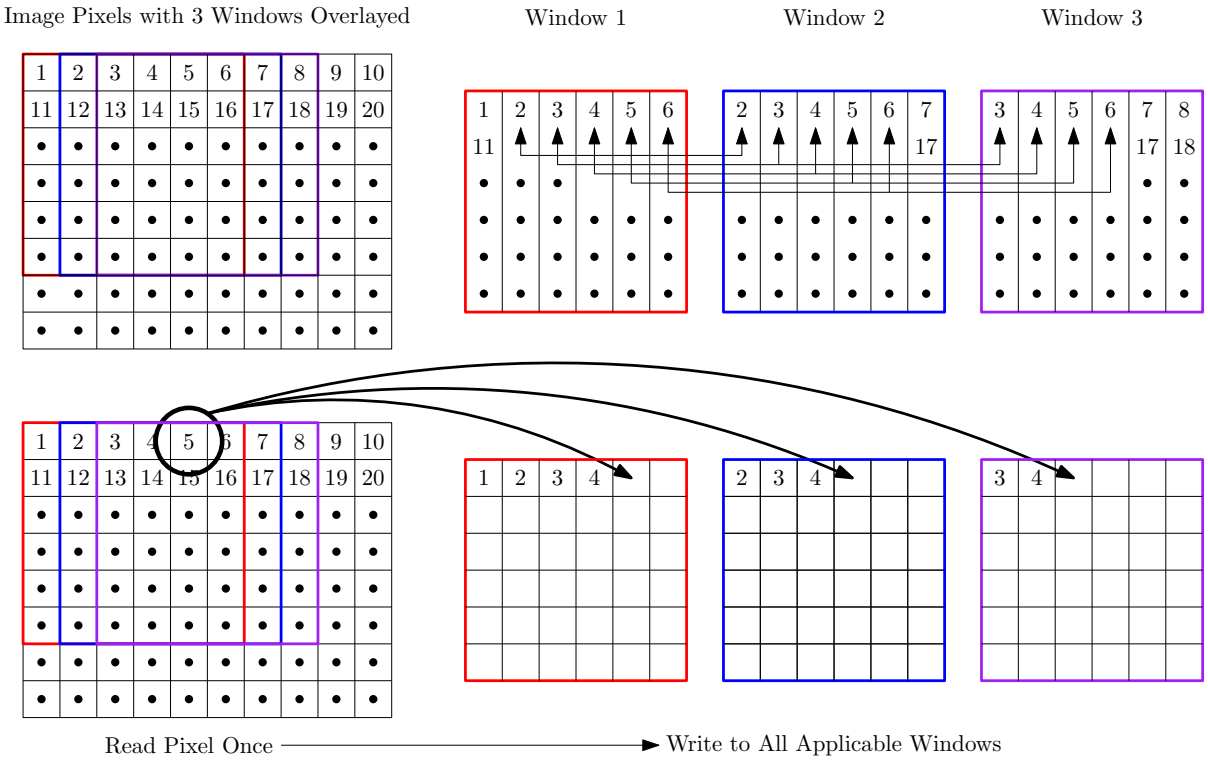


Figure 3.6: Loading One Pixel into Multiple Windows

that ideally in 8 reads from the integral image, the entire first row could be loaded into all three windows. If this is done for all 6 rows, then all three windows could be loaded with only 48 reads from the integral image required. In contrast t $6 * 6 * (3\ windows) = 108$ reads if these were loaded independently.

Another way the window caches can be loaded more rapidly is by structuring the on-chip memory for the integral image so that multiple pixels can be read every cycle. This can be accomplished by making the integral image memory wider. Figure 3.7 illustrates how this would improve window cache loading performance. The number of pixels that can be read per cycle, referred to as the *PixelRate*, is a compile-time parameter of the system.

We can combine these two techniques and load multiple windows simultaneously from integral image reads of many pixels per cycle. This is shown in Figure 3.8. When the on-chip memory is structured so that multiple pixels can be read from the integral image

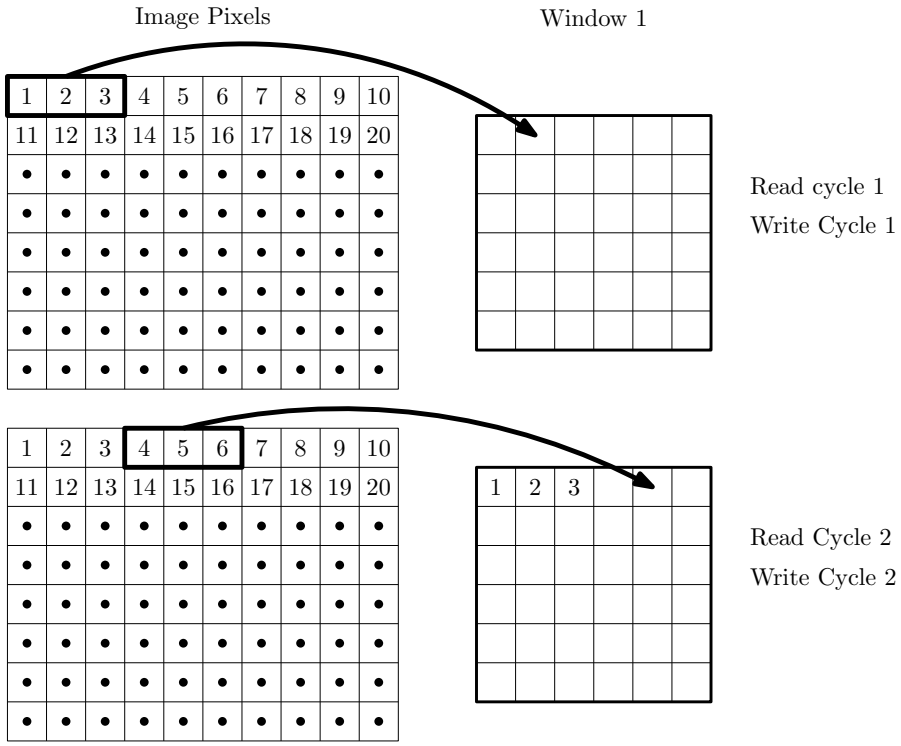


Figure 3.7: Loading Multiple Pixels into a Single Window

per cycle, multiple pixels must also be written to the window caches per cycle. Figure 3.8 illustrates that to write the first block of data to each window the pixel data from two reads is needed. Each block that is written is a variably shifted version of that data. This variable-shifting will be needed for the window caches associated with each PE. The hardware block that does this denoted window alignment is illustrated in Figure 3.9. The amount of alignment each alignment block can shift data is equal to the number pixels read per cycle which is a parameter of the system we call *pixelRate*.

The implementation of the window alignment block avoided using costly mux trees or barrel shifters in lieu of a series of fixed registers. The exact connections between the registers is controlled by an offset parameter which is used at the time the hardware is synthesized. The result seen in Figure 3.9 is a very simplistic block which can accomplish the need to group write together for each window cache.

The complete window cache loading hardware is shown in Figure 3.10. It exploits the similarity of neighbouring windows in order to load them at a much faster rate than

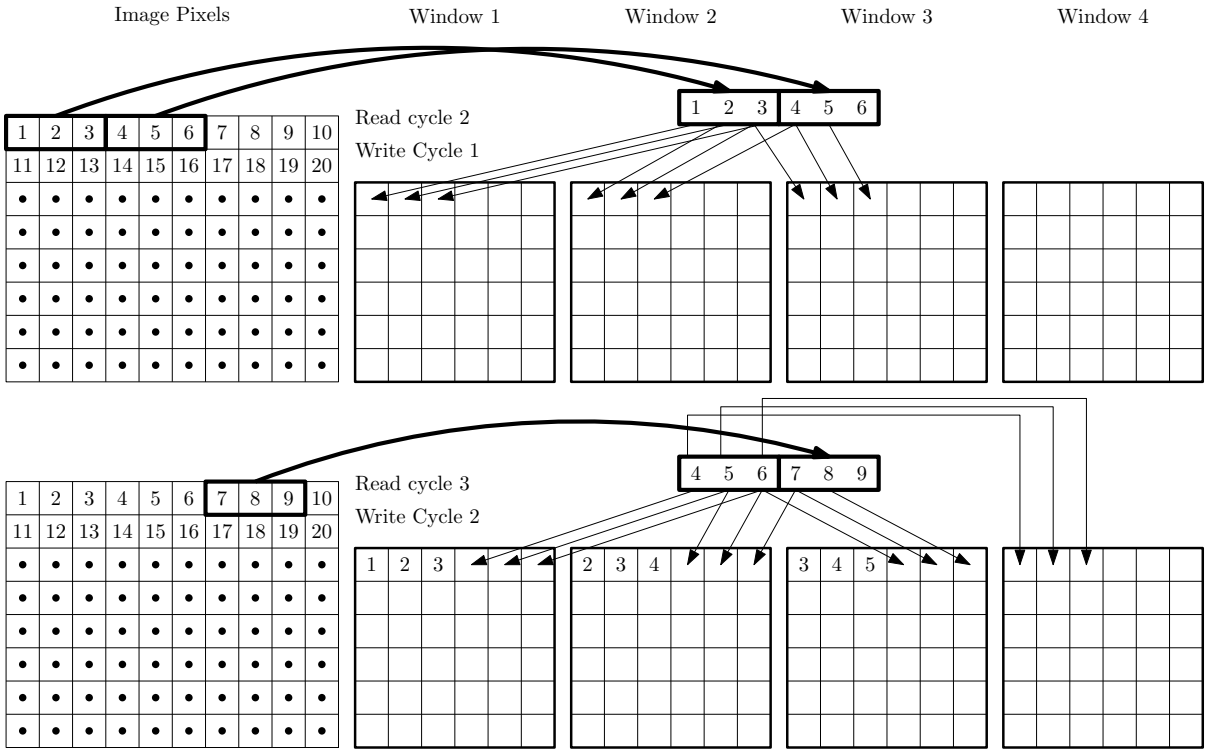


Figure 3.8: Loading Multiple Pixels into Multiple Windows

in previous works. As a quick example, consider a system with a window size of 20x20, 24 *Cores*, and a *pixelRate* of 8. It would take 6 reads to fill the first row of all 24 cores and this would be done for each of the 20 rows. That is a total of 120 cycles to fill 24 windows or an average of 5 cycles per window. Loading them independently would take 400 cycles per window. As a result of our system, we could ideally achieve an effective on-chip memory bandwidth increase of 80 times. That gain came without using more memory bits than previous work with the same number of window caches.

3.2.2 SIMD Processing

In our SIMD paradigm, the data from the cascade of classifiers can be considered our instructions as they tell the hardware processing elements what data should be fetched from their associated 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

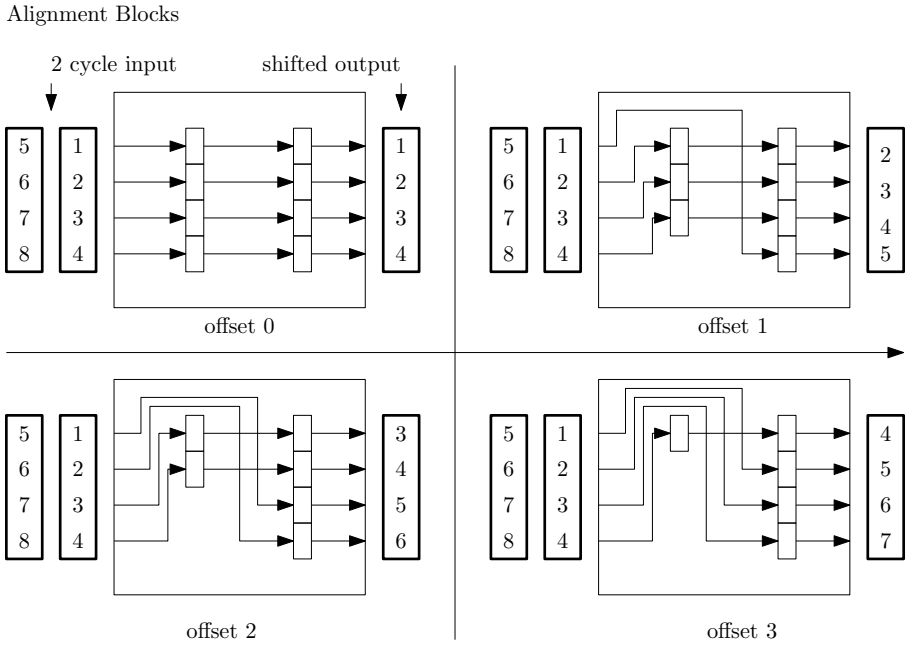


Figure 3.9: Simple Alignment Hardware with Various Offsets

the cascade of classifiers as seen in Figure 3.11. The data that this memory returns is broadcast to all PEs which simultaneously process their window caches rectangle by rectangle, feature by feature, classifier by classifier. Recall from Chapter 2 that a window being processed can potentially fail at the end of every classifier stage. 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 broadcast data from deeper in the cascade memory and thus prevent PEs that are finished from starting again until every PE wants to start again.

The hardware system and the two key architectural innovations of this work have been presented with a focus on performance. In the next section the focus will be on the considerations and choices made to ensure the hardware system produces accurate results.

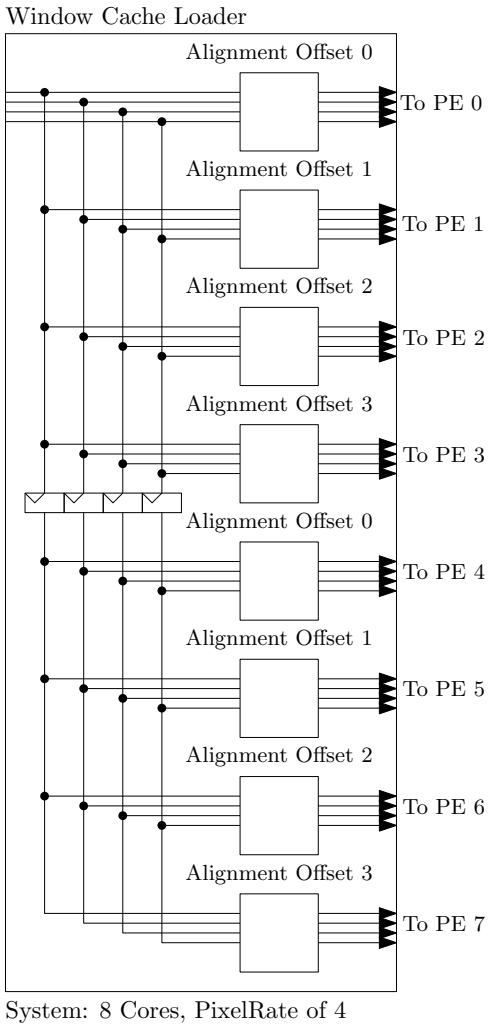


Figure 3.10: Parallel Window Cache Loader

3.3 OpenCV Compliance

Recall from Chapter 1 that the definition of OpenCV compliance is, given the same input stimuli, the output of the software and hardware version of the MSHCD algorithm should be the same. A minor modification to the OpenCV source code that was made however, in order to achieve this compliance. We acknowledge the inconsistency of claiming to match the results of OpenCV only after modifying the source code. In this Section will explain the justification and nature of this modification which addresses needless fixed point errors during image scaling and does not modify the core object detection code.

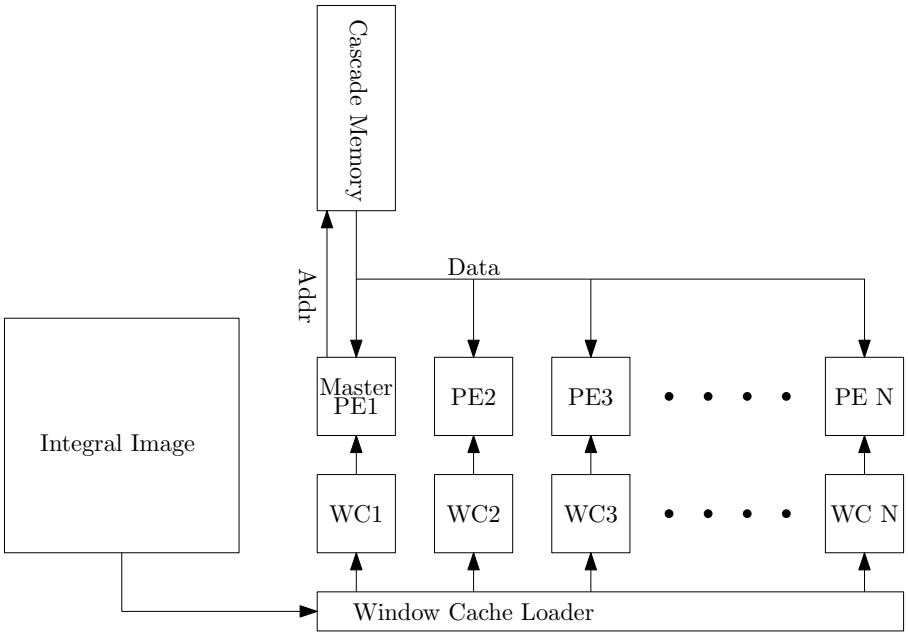


Figure 3.11: SIMD Haar Detection Architecture

3.3.1 OpenCV Scale Modification

When an image is scaled as part of the MSHCD algorithm the first step is to calculate the exact pixel dimensions of the new smaller image. Recall from Section 2.5.1 that these dimensions are calculated in the following way $scaleFactor^i * origonalImageSize$, where i is $0, 1, 2, 3.....$ Using floating point the $scalFactor^i$ can quickly saturate the available representation bits in a float; this is shown in Table 3.1.

ScaleFactor	Iteration	Result
0.85	0	1
0.85	1	0.85
0.85	2	0.7225
0.85	3	0.614125
0.85	4	0.52200625
0.85	5	0.443705312

Table 3.1: ScaleFactor Representation Bit Increase

As a result of this saturation we can be sure that there will be fixed point rounding errors on the FPGA which can lead to an incorrect calculation of the scaled image size.

This error can cause the scaled image dimensions to be rounded to the wrong whole number. The error gets compounded as the scaled image size is used to calculate weights and location indices into the original image, which can now both be wrong. These two values are then used to calculate the new value of the scaled image pixels which can now be wrong. Those incorrect pixel values will get summed to form the integral and square integral images which will be incorrect. The windows from the integral image will be fed into the main processing elements. By this point it would be difficult to verify that the hardware and software were performing identical computations as the input windows in the two cases would be significantly different. At the same however the scaled images in both software and hardware could have pixels that differ by at most 1 and would be indistinguishable to a human.

To address this issue and make sure that the input windows processed by the PEs in hardware and the inner loop of the software were exactly, bit for bit, identical, a small modification was made to the OpenCV source code. The code that performs image scaling was converted from floating point to 36 bit fixed point, which is the same representation as in the hardware. Doing this ensures that any computation near the boundary point of of a Round() operation would behave the same in both software as well as hardware. By doing this it guarantees our results for each scale up to and including the integral images *exactly* match the software version. The computation performed by the PEs on their window caches, the core of the object detection algorithm, do not suffer from accumulating fixed point errors. In fact the next section will show how the rest of the MSHCD algorithm actually dampens fixed point errors.

3.3.2 Window Processing Fixed Point Errors

We previously showed how fixed point errors in image scaling get compounded with successive steps of the MSHCD algorithm, and thus eliminating those errors was important. Here we will show how fixed point errors that occur while processing individual windows

of the integral image are dampened with successive steps of the MSHCD algorithm, and thus we do not need to do any more modification of the OpenCV software.

As discussed in Section 2.5.3 a feature is applied to a window by multiplying each of the feature rectangles by a weight, given as part of the cascade of classifiers. The products are added together into a feature sum and compared against a threshold value for the feature, also given as part of the cascade of classifiers. Depending on the result either the known value $alpha1$ or $alpha2$ is added to the classifier sum; one *alpha* value is added for every feature in the given classifier. In the case that the feature sum is very close to the feature threshold, it is possible for a fixed point error to cause the wrong *alpha* to be added to the classifier sum. However, this can only affect the classifier if the classifier sum would have already been close to the classifier threshold and the addition of an incorrect *alpha* caused the classifier stage to incorrectly report a pass or fail.

If a classifier stage passes when it was not suppose to pass, the output is only affected if that window then went on to pass all other classifier stages of the cascade of classifiers and declare that an object was found incorrectly. Alternately, if a stage fails when it should have passed, the output is only affected if the given window would have gone on to pass all the remaining classifier stages and report an found object but now due to this error did not.

Given an image that contains 3 objects one would expect a clustering of 4-5 positive results for windows near each object. For these fixed point errors to case a false negative they would have to cause each of the clustered positive results for a single object to fail prematurely. On the other hand one would only expect 15 or so windows to pass the whole cascade out of a total of hundreds of thousands of windows in the image. Any particular window that gets one classifier further in the cascade due to a fixed point error would almost certainly fail the later classifiers simply due to the fact that so few windows can pass the later classifiers.

3.4 Summary

In this Chapter a detailed description of the MSHCD hardware system was presented. This included the two key architectural innovations of this work, parallel window loading and SIMD window cache processing. Finally, the management of fixed point arithmetic errors was discussed. In the following chapter the performance and energy usage of both the software and hardware version of the MSHCD algorithm will be explored.

Chapter 4

Results

In this Chapter we will present measurements of the MSHCD system described in Chapter 3 and compare these measurements to alternative computational media including a mobile phone processor. The Chapter will begin with an outline of our experimental methodology and then discuss results that were obtained and their significance in the context of mobile devices and mobile FPGA co-processors.

4.1 Experimental Methodology

In this section we will present the technical specifications of the platforms on which the MSHCD algorithm was implemented, the input image set and the OpenCV cascades that were used, our methodology for measuring performance and power, and which configurations of our FPGA system were used.

4.1.1 Hardware System Description

The OpenCV MSCHD algorithm was run on three machines as part of this work: a Desktop, a mobile phone, and a FPGA.

1. 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.

2. 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.

3. FPGA: The FPGA used was a Stratix IV GX EP4SGX530 which has 531K LUTs, 27Mbits of on-chip memory and 1024 18x18 multipliers [29]. The FPGA is built on a 40nm CMOS process technology and was available in the second quarter of 2010. The FPGA is part of the Terrasic DE4 Development board which is an educational development platform. 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 clocked at 125MHz.

Notice that although all three computational medium were built on different process nodes they were all released at roughly the same time and were considered high end devices at the time of launch.

The Desktop machine will run the MSHCD algorithm based on OpenCV 2.1 with all available OpenCV performance optimizations enabled when the library was compiled which includes Intel specific optimizations. The mobile phone also ran the MSHCD algorithm based on OpenCV 2.1 but the library was compiled using the Android NDK into android binaries as described in Section 2.2.2 and were linked to from the Java application environment of Android. Compiling directly to object code with the NDK was very important due to the performance benefits described in Section 2.2.2 and shown

in Table 2.1. For the mobile processor, most hardware performance optimizations such as Intel SSE vector instruction support had to be disabled as they are not supported on the ARM hardware of the Nexus One. I did not rewrite the code to support ARM neon vector instructions.

4.1.2 Input Images and Cascades

In this Section we described the input stimulus (images and classifier cascades) used to measure the object detection algorithm on all platforms. The selection of a classifier cascade can have an effect on absolute run-time of the object detection algorithm as larger more complex classifier cascades can result in slower performance. The extra computation associated with a larger cascade will exist on all platforms and thus will not affect the relative performance between them.

The OpenCV release comes with 19 classifier cascades [15] 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. There are a few distinct cascades which will detect each of these human parts. Of these cascades, three were selected to be used in our measurements. These cascades were also used in the OpenCV sample application demonstrating the use of MSHCD algorithm. 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 4.1. As with all Computer Vision, OpenCV is not perfect and it missed the detection of one eye. Table 4.1 shows the number of features per classifier stage as well as the total number of features and stages for the three cascades that were chosen.

The selection and number of input images can be important when testing object detectors for the quality of their results, which is not dependant on execution time but

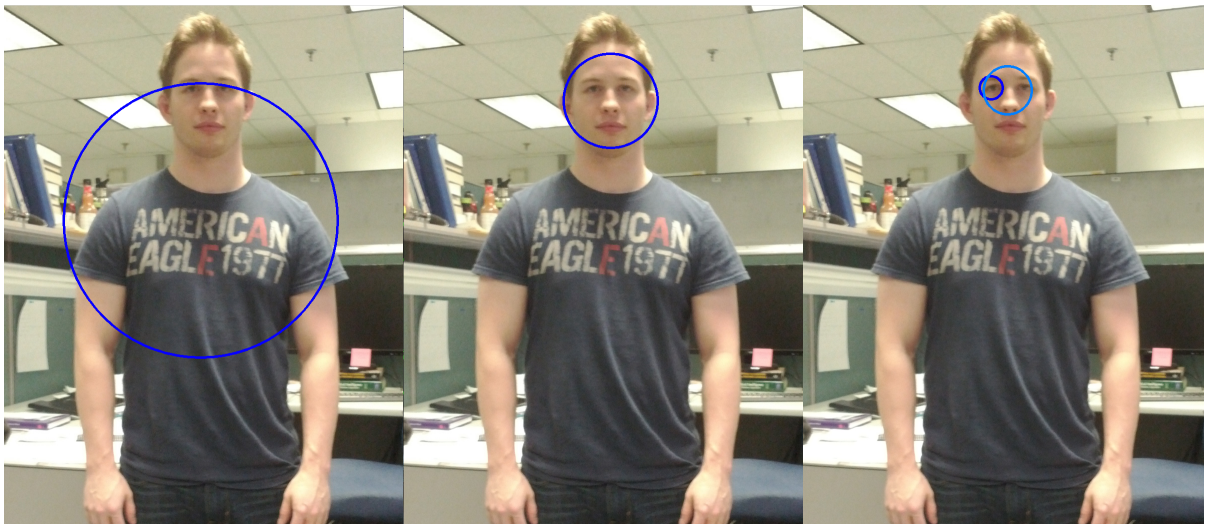


Figure 4.1: Object Detection

Haar Features per Stage			
Stage\Cascade	Eye	Upper Body	Face
1	6	20	3
2	12	33	16
4	16	42	39
8	36	43	51
12	32	61	103
16	51	88	137
20	69	101	182
Total Stages	24	30	22
Total Features	1066	2432	2135

Table 4.1: Haar Features per Stage of Chosen Cascades

rather on the rates at which the implementation correctly and incorrectly detects objects. However, our work by design produces the same results as OpenCV so the quality of results is not in question. If the OpenCV implementation produces results of high enough quality for a particular application then this work can be used to accelerate that computation. The goal of this work is to accelerate the computation relative to software, and use less overall energy for the same work.

The speed of the MSHCD algorithm is dependent, among other things, on the input image. The more 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.

As a result of these considerations, we chose an image set of 10 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 MSHCD algorithm has been commonly used at image sizes as small as 160x120. For this reason each of 10 images we chose were tested at 10 sizes ranging from 320x240 to 120x90 for a total of 100 images in the test set.

4.1.3 Measurement Methodology of Performance and Power Consumption

Two key metrics of the object detection system are performance and energy consumption. In this Section we describe the methodology behind these measurements.

Performance

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, their location and their scales. This time measurement is a function of the input image, cascade and also the scale factor described in Section 2.5.1. 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. Using OpenCV as a guide, the scale factor is set for all our experiments to the value used in the sample MSHCD code of 1.1 (10%).

The USB1 data connection that was implemented between the FPGA and host system is not representative of an how we envision a connection from a processor to an embedded

mobile FPGA co-processor. In that context a shared memory interface or high speed internal bus would be expected, either of which would be significantly faster than USB1. To stream a 320x240 8-bit image at 30fps requires a 19 Mbit connection, and even in a low power mobile context, having a connection to the FPGA this fast is a reasonable expectation. For this reason we do not include the data transfer time to and from the FPGA system and measure only the time to perform the MSHCD computation.

For the desktop and mobile computation media, the time is measured around the function call to the MSHCD algorithm after the cascade and input image have been loaded from disk into memory and initialized.

Power

Power was measured only on the FPGA and the mobile phone as this was the main focus of this work. The power consumed by our desktop processor was only roughly estimated as it is well over an order of magnitude greater than would be acceptable in a mobile context.

To measure the power used by a mobile phone a percentage discharge of the battery was recorded while the phone was idle and while performing the MSHCD algorithm. The Nexus One has a 1400mAh battery that operates at 3.7V. With this information power can be calculated with a few simple operations.

$$total\ mAh\ used = 1400mAh * \%\ Battery\ Used$$
$$average\ mA = (total\ mAh\ used)/(hours\ taken\ to\ discharge\ battery)$$
$$mW = (average\ mA * 3.7V) * 90\% power conversion efficiency$$

In contrast to the mobile device, the power of the FPGA system was not determined by directly measuring the power consumed by the FPGA while it was running. As discussed in Section 4.1.1 the FPGA used is part of a large educational development board, the DE4. Most of the components on this board would not exist in the embedded

mobile context in which this work is interested. Instead, the power of the FPGA was *estimated* using software tools provided by the vendor of the FPGA; Altera. Power estimation can be done with varying degrees of accuracy, with the most accurate based on a timing simulation of a circuit's operation.

The Altera power estimation software 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, the tool can generate an accurate estimation of the overall power consumed by the FPGA. However, timing simulations of large systems can take a prohibitively long time. To avoid this only a 5ms timing simulation was performed on our system. A 2ms slice at the end of simulation was used to estimate the overall FPGA power usage. This 2ms slice occurs during the SIMD processing of windows from the first image scale and is representative of the highest power usage that can occur while the algorithm is running. Section 4.4.2 will show that the PEs consume a significant amount of the total FPGA resources used in the system, which is why our power measurements occur at a time when the PEs are actively computing,

4.2 The Measured System

This Section begins with an outline of the various parameterizations of the MSHCD hardware that was tested. The speed of the hardware, desktop and mobile device are presented and analyzed. The FPGA resource utilization is then presented to show the cost of achieving the performance previous discussed. Next the power and energy per operation results are shown. Finally the MSHCD hardware from this work is compared to the most similar previous work.

Hardware Parameters Explored

The maximum image size was set to 320 pixels wide 240 pixels high. (A larger resolution, such as a VGA resolution of 640x480 results in a prohibitively large on-chip memory usage due to the size of the integral and square integral images. A solution to this problem will be discussed Chapter 5.) The maximum number of features in the cascade of classifiers was set to 3500. This number was selected so that the hardware cascade memory would be large enough to store any of the 19 classifier cascades that come with OpenCV source code. The maximum window size was set to 30x30 and again this was so that the hardware system would work with every classifier cascade that came with OpenCV. The scaling factor, as mentioned before, is set to 10% for all experiments.

In our experiments we will vary the the number of SIMD PEs or *cores* as well as the *pixelRate*, defined in Section 3.2.1: the *pixelRate* is the number of pixels from the integral image that can be loaded into a window cache every cycle. The number of *cores* will range from 1 up to 40 and the *pixelRate* from 1 up to 8 where applicable. The value of *cores* must be an exact multiple of the *pixelRate* so some of these configurations are invalid.

4.3 Functionality

In accordance with our definition of OpenCV compliant the results of the MSHCD algorithm running on the desktop computer, mobile device and FPGA were the same. The location and scale of each detected object in each image used in this work was equivalent across platforms. Recall that the desktop and mobile device received a minor modification to OpenCV that was described in Section 3.3.

4.4 Performance

In the following section experimental results of each of the system variations are presented. This includes a performance comparison of the OpenCV MSHCD algorithm running on each of our platforms, power and energy usage results, the FPGA resource utilization of the hardware system for various configurations, and finally an overall comparison to the most relevant previous work.

4.4.1 Speed Up

Here the computational performance of the OpenCV MSHCD algorithm is presented. First, we will show the relative performance of our hardware across the variations described above. Next we will compare the best hardware configuration against the desktop and mobile devices described earlier.

Performance Across System Variants

Our first experiment measures the impact of the *pixelRate* and number of *cores* on the performance of the system. In Section 3.2.1 it was show how increasing the *pixelRate* decreases how long it takes to load the next set of window caches. This will increase the performance of the system as long as there are still situations where the PEs have collectively finished their workload before new data has been loaded.

The performance was measured across the 100 input images (described in Section 4.1.2) for each of the three chosen classifier cascades. Figure 4.2 shows the relative performance increase of the system, normalized to a *pixelRate* of 1, as a function of the number of cores in the system. Note that the number cores must be an exact multiple of the *pixelRate*, which is why some bars appear to be missing.

The overall system performance increase in having the ability to read multiple pixels per cycle from the integral image is more than 2 times. We notice that increasing the

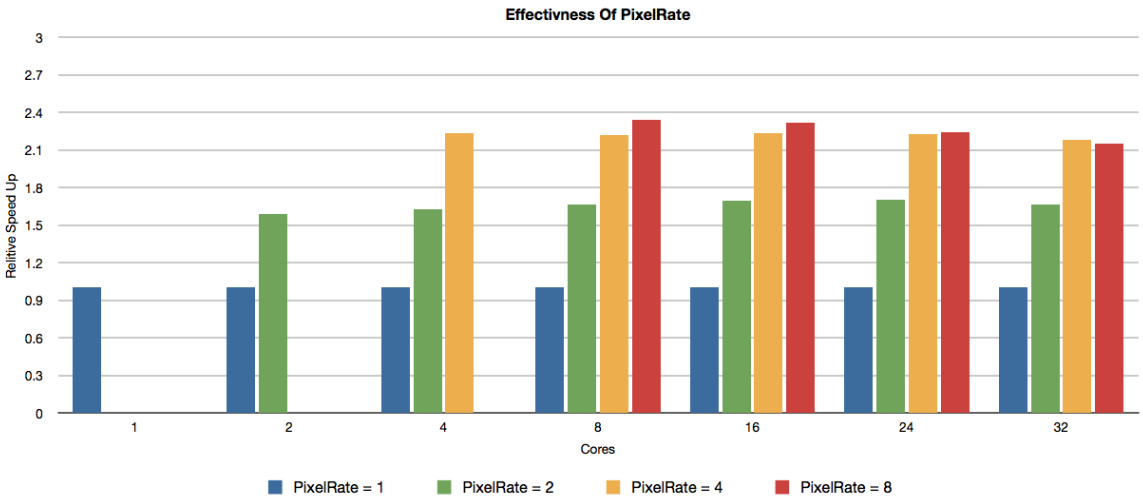


Figure 4.2: Effectivness of PixelRate

pixelRate to 8 has very little effect on overall performance. 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, as we will show in Section 4.3.2, comes with an extremely small resource overhead.

Next we look at how well the system performance scales as the number of *cores* is increased. Figure 4.3 shows the relative performance increase of the hardware variants normalized to the slowest system; one core and a *pixelRate* of 1. Each curve on Figure 4.3 represents different *pixelRate*. Table 4.2 shows the raw performance data for the subset of input images that had the largest dimensions, 320x240.

Figure 4.3 shows consistent but sub-linear scaling up to approximately 32 cores, at which point the performance improvements become negligible. The largest speed up using a *pixelRate* of 4 and 32 *cores* is 18.4 times.

As the number of cores increase, the number of windows being processed in parallel increases as well. As this happens the windows contain image data that becomes decreasingly correlated, a visual example of this is shown in Figure 4.4. When this happens the SIMD PEs become less efficient as more often PEs that failed at an early stages will be stalled waiting for other PEs to finish.

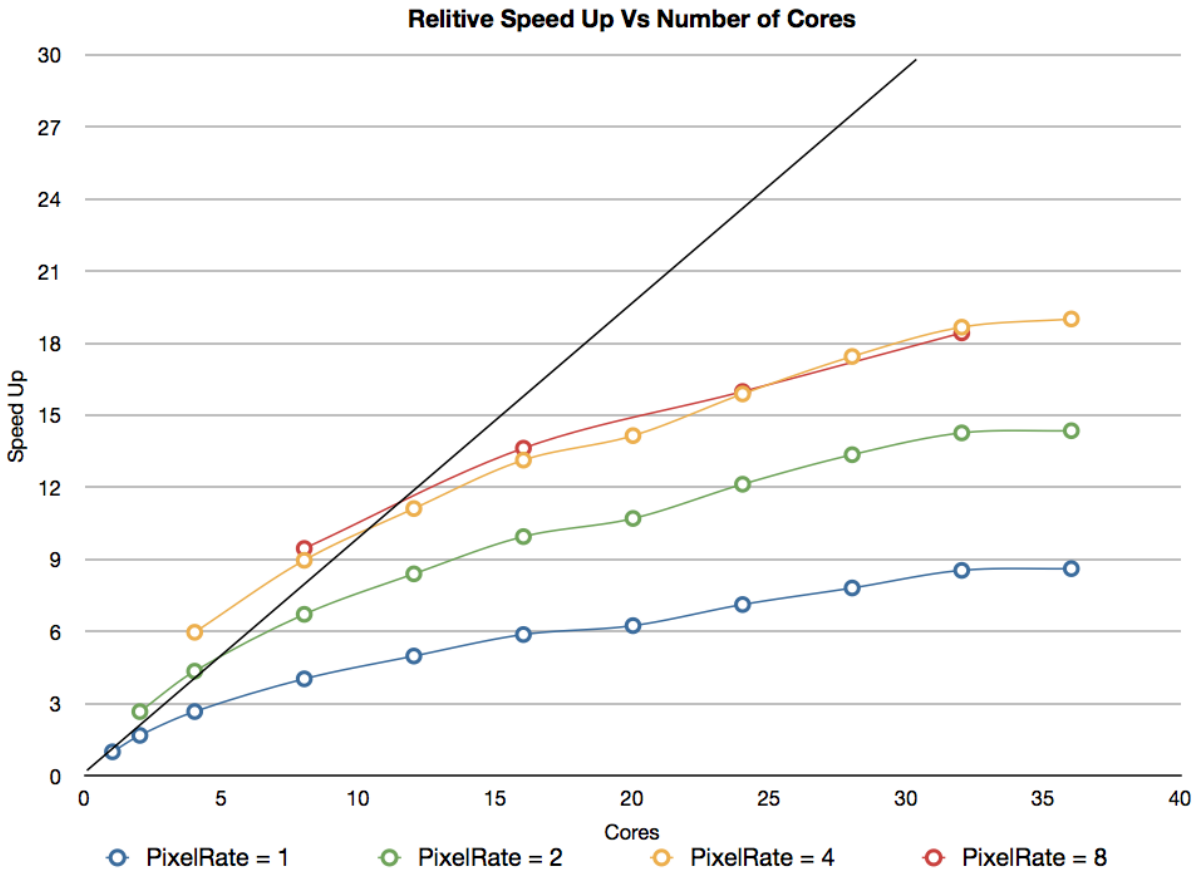


Figure 4.3: Performance vs Number of Cores

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
12	76	45	34	NA
16	65	38	29	28
20	61	36	27	NA
24	53	31	24	24
28	49	29	22	NA
32	45	27	20	21
36	44	27	20	NA

Table 4.2: Execution Time per Frame for Each Hardware Configuration

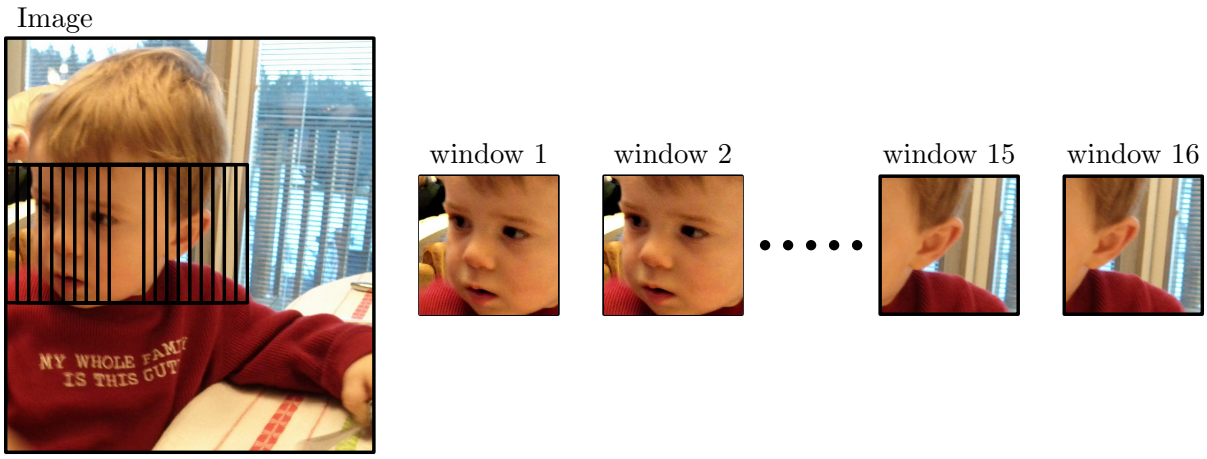


Figure 4.4: Distance Windows

Performance Comparison to Processors

For comparison with other platforms, the hardware configuration of 32 *cores* with a *pixelRate* of 4 will be used. Increasing these parameters above these values does not add any significant performance increase. Table 4.3 gives the absolute run-times and relative speedups of the hardware system when compared with the mobile device and the high end desktop computer both running the OpenCV 2.1 software implementation of MSHCD for the same set of images.

The desktop version (as described in Section 4.1.1) has been well optimized including Intel SSE vector instructions support but is not multi-threaded. Hardware level optimizations were lost when OpenCV was compiled with the NDK for Android. This puts Android at slight disadvantage in that the ARM NEON vector instructions are not being utilized.

Performance Comparison			
Metric	Mobile Device	Desktop	Hardware
Runtime (ms)	1180	77	20
Relative Speed Up	**1**	**15.3**	**59**

Table 4.3: Platform Performance Comparison

The FPGA implementation outperforms the mobile device, by a factor of 59 times.

While the mobile device takes more than a full second per image, the hardware implementation can run, in theory, at 50 images per second, which is more than fast enough for real-time video.

In a practical Computer Vision application it is unlikely that a full object detection algorithm would be run on every frame. After the object detector has been run a much more localized and computationally less intense tracking algorithm would be run to follow that object's small movements over the next several frames. Given a mobile device with an FPGA co-processor this core could easily compute a full object detection every, for example, 5 frames. This would free the otherwise 100 % busy CPU to interpret and make creative use of the results in an interesting application, while still being very confident about the locations of objects and using less power than if the FPGA was to perform the object detection every frame.

Somewhat unexpected was the 4 times performance improvement the hardware had over the high end desktop. This computer was running a 6-core 32nm processor clocked at 4.0Ghz but the OpenCV MSHCD is not multi-threaded so only 1 of the cores is being utilized. If the OpenCV implementation included a robust multi-threaded implementation we could expect the performance gap between the hardware and software to narrow but for the FPGA to still consume significantly less power.

4.4.2 FPGA Resource Utilization

In this Section we measure the FPGA resources it took to achieve the performance described above. We will look at the four main FPGA resource types: LUTs, registers, on-chip memory bits, and DSPs blocks as described in Section 2.2.3.

Register Usage

The usage of registers vs number of *cores* with a *pixelRate* of 4 is shown in Figure 4.5. In a 36 core system 86,665 registers are used.

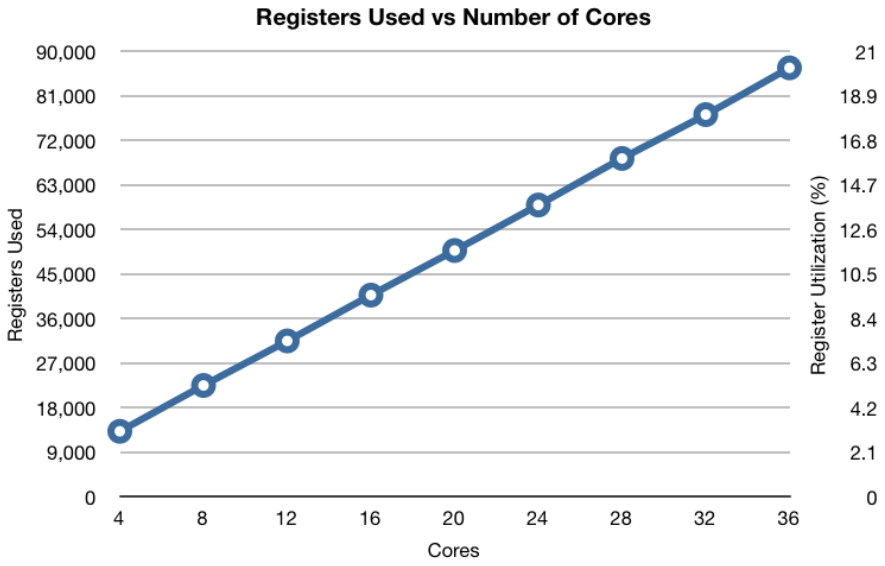


Figure 4.5: Register Utilization

Increasing the *pixelRate* actually decreases the total register usage of the system, as shown in Table 4.4, albeit by a marginal amount. The net effect, approximately no

Impact of PixelRate on Registers in a 32 Core System				
Resource	pixelRate = 1	pixelRate = 2	pixelRate = 4	pixelRate = 8
Registers	78170	77544	77219	77386
Relative Increase	1	0.992	0.987	0.990

Table 4.4: Impact of PixelRate on Registers

change, is a result of two competing effects. Pixels read from the integral image are registered in several places as the pixel is copied, aligned and prepared to be loaded into multiple window caches. When the *pixelRate* increases the number of pixels read from the integral image increases and the size of all of these intermediate registers increase. When the *pixelRate* increases there are also fewer addressable locations in the integral image, each of which store more data. If the number of address bits required to access the integral image decreases, so too does the size of every register in the system that has to store, count or compare a value that depends on the integral image address.

aLUT Usage

An aLut is the core bock of hardware n an FPGA that performs combinational logic. The usage of aLUTs is shown Figure 4.6, the maximum number of aLUTs used in a 36 *core* system is 67,172.

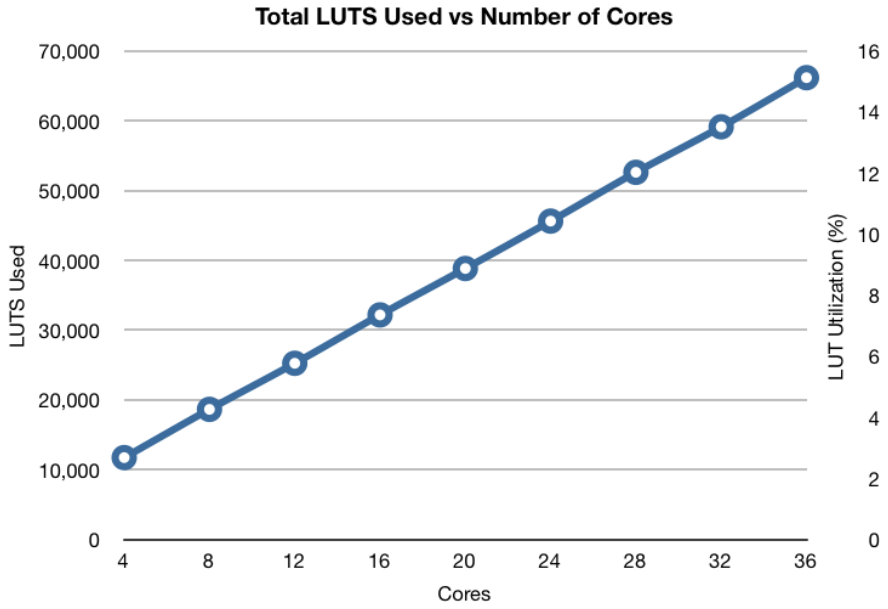


Figure 4.6: Lut Utilization

The effect of increasing the *pixelRate* initially decreases the total aLUT usage of the system and then proceeds to increase it, by small amounts as seen in Table 4.6.

Impact of PixelRate on LUTs in a 36 Core System				
Resource	pixelRate = 1	pixelRate = 2	pixelRate = 4	pixelRate = 8
LUTs	60154	58518	59133	60577
Relative Increase	1	0.973	0.983	1.01

Table 4.5: Impact of PixelRate on LUTs

The aLUT usage initially decreases as the *pixelRate* increases because the number of address bits needed to represent the integral image and window caches goes down. All of the arithmetic operations that convert (x,y) integral image and window cache coordinates into a one dimensional address, decrease in size. Competing with this effect is the cost

of selecting a single pixel from a multiple pixel read, which increases with the *pixelRate*. Window caches are written to multiple pixels at a time, but the variance caches are not. Logic is needed for every PE to select a single pixel at a time to write to the variance cache. The net effect on aLUT usage for the system configuration we experimented with is negligible.

DSP and Memory Usage

The usage of DSPs in our system is much more limiting than registers or logic. A chart of DSP usage as a function of the number of cores in a system of *pixelRate* 4 is shown in Figure 4.7. The DSP utilization climbs to 544 18-bit DSPs for a system with 32 cores, with each core needing 14 18x18bit DSPs.

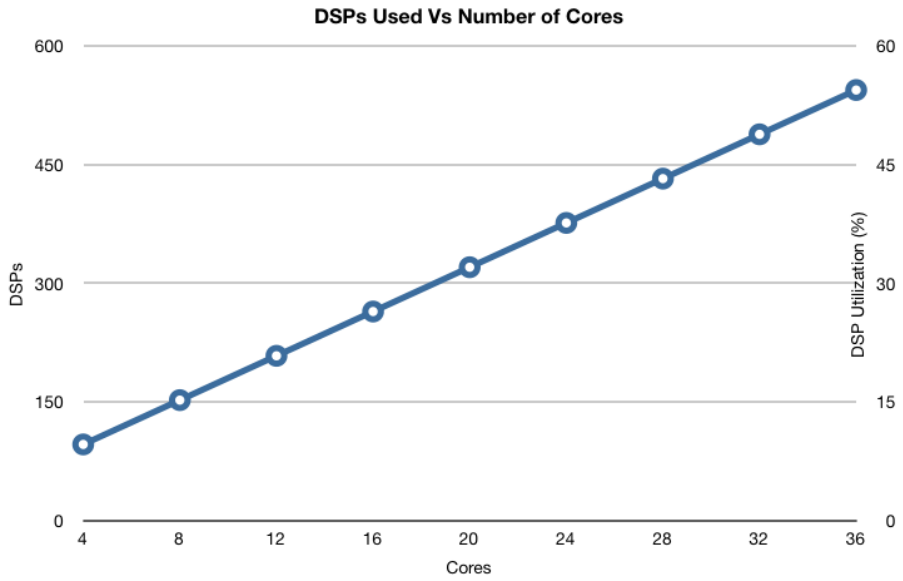


Figure 4.7: DSP Utilization

Next we will discuss the impact of on-chip memory usage on the system. Figure 4.8 shows how the on-chip memory usage scales with an increasing number of cores in the system. The basic use of the on-chip memory is very high as our system requires storage for 4 image memories and a large classifier cascade. The cost of scaling the number of cores, which amounts to the cost of 2 additional window caches, is actually quite low.

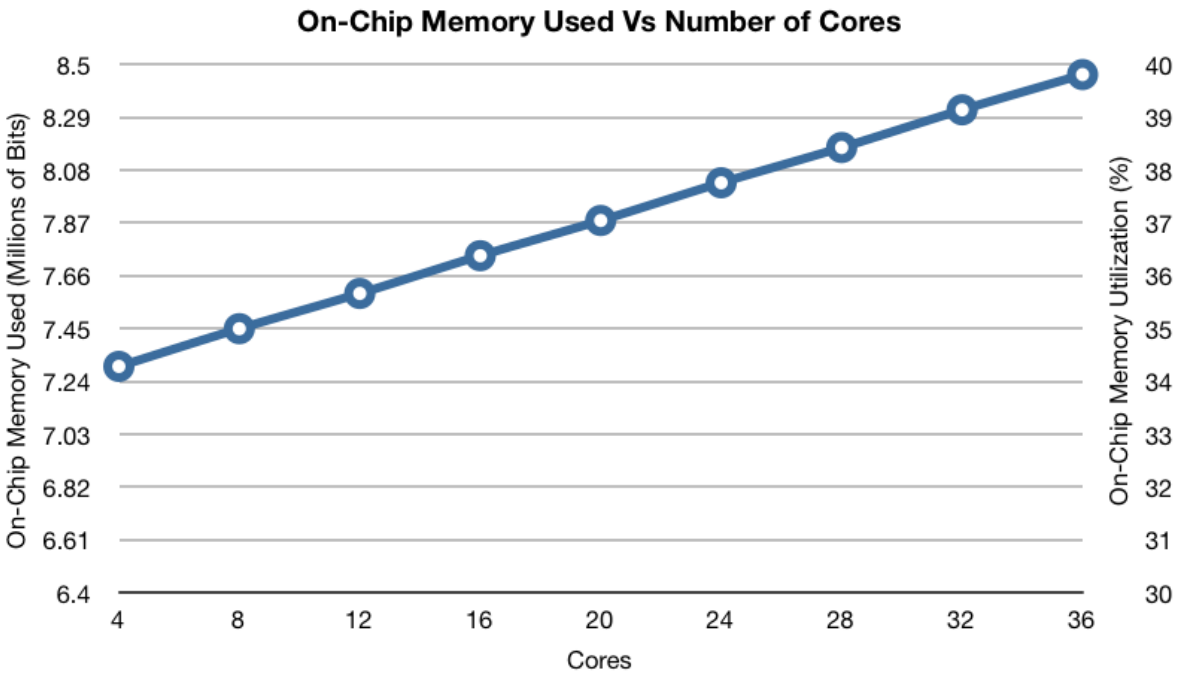


Figure 4.8: On-Chip Memory Utilization

Each additional core uses 3.6k memory bits. This means that even on a smaller FPGA having a considerable number of processing cores is feasible if the initial on-chip memory utilization is lower. There is known technique for substantially reducing the on-chip memory usage of the integral and square integral images that is not implemented in this work but has been implemented in previous works including [16]. This will be discussed further in Chapter 5.

4.5 Power and Energy Consumption

In this Section we discus the power and energy comparison of the mobile device and the FPGA system running the OpenCV MSHCD algorithm. We will first present the power usage of each system and then the energy used per computation of an input image.

Table 4.6 and Table 4.7, shown below, summarize the power results for the mobile platform, the desktop and the FPGA platform. The desktop cpu has a maximum rated power of 130W and a conservative estimate the lower bound actually power usage is 60W.

The battery used percentage of the Nexus One was recorded while the phone was idle with the screen on and all radios disabled for an extended period of time. It was also recorded and while the phone was running the MSHCD algorithm. The FPGA power was estimated using a timing simulation as described in Section 4.1.3.

Mobile Power Consumption							
Mobile Device	start %	end %	time (s)	battery % used	mAh used	average mA	power (mW)
Idle	91	85	4355	6	84	69	231
Running	84	44	6659	40	560	303	1008
Running - Idle				34	476	233	777

Table 4.6: Mobile Power Consumption Running OpenCV MSHCD

FPGA Power Consumption			
i/o power (mW)	static power of entire FPGA (mW)	dynamic power (mW)	total power (mW)
194	1510	1699	3402

Table 4.7: FPGA Power Consumption

The increase in power usage by the Nexus One as a result of computing the MSHCD algorithm was 863mw and the total estimated power used by the FPGA implementation was 3402mw. Using this data both the power usage and computation speed for the Nexus One and Stratix 4 FPGA have been presented in this Chapter. With these two values the energy efficiency of each platform can be computed. Energy efficiency is measured by the total amount of energy needed to perform one the MSHCD per input image.

To be as conservative as possible, only power differences between running and idle for the Nexus One was included in energy efficiency calculations, while the total power of the FPGA was included. The reason for this is simple: the amount of the power required for the phone to be turned on and running the operating system is a baseline needed before any applications can start. This amount of power (231 mW from Table 4.6) would still be needed even if the MSHCD was running on the FPGA co-processor. On the other hand the FPGA could be completely turned off if it was not being used.

Table 4.8 shows that the hardware system uses roughly 4.5 times more power than

Energy Per Computed 320x240 Image				
Device	power (W)	time per image (s)	energy per image (j)	relative energy per image
Nexus One	0.777	1.18	0.916	**1**
Intel 980x max	130	0.077	10.01	**10.9**
Intel 980x estimated	60	0.077	4.62	**5.0**
Stratix 4 EP4SGX530	3.402	0.020	0.068	**0.074**

Table 4.8: Energy per Operation

the mobile device but because it can can perform the computation nearly 59 times faster it is 13.5 times more energy efficient. This is a crucial result for a number of reasons. Mobile devices are designed around power efficiency, it is well understood how to make a faster processor given a larger energy budget. Smart-phone processors must be designed to be as fast as possible while still lasting a full day of usage on a single charge. If our system was much faster than the mobile device but needed a proportional increase in the total power consumed it would not be of practical relevance. It is also an important result given how conservative the result is. The Stratix 4 FPGA is designed with high clock speeds and large designs in mind. Even our 32 cores system would only require half of this FPGA and run on a relatively modest 125 mhz clock.

4.6 Comparison to Previous Work

Outside of the mobile context, others have worked to accelerate versions of a Haar object detection algorithm to various degrees on an FPGA. The work that most closely resembles the present work's focus on practical cascade and images sizes as well its incorporation of image scaling was Hiromoto [16] and was discussed in Section 2.6.3. Recall that [16] processes one window at a time. However for that window every feature in a fixed number of stages is computed completely simultaneously. If the window passes all of those stages a slow serial process is invoked to check each of th subsequent stages until the window fails (or an object is detected). Windows that would fail before the serialized hardware is invoked will finish quickly while windows that would reach a late stage of the cascade

will take a very long time to finish.

The comparison between our performance and resource utilization with [16] must be approximate because the same cascade and images are not used. They use a 25 stage cascade that is similar to OpenCV cascades in terms of features per stage. Table 4.9 shows a comparison of the features per stage of an average OpenCV cascade and the cascade used in [16]. The OpenCV cascades have a few more features in their earlier stages but seem to grow at a similar pace. Although the two works did not use the same cascades they are close enough that a direct performance comparison will be valuable even if it is only a first-order approximation.

Features Per Stage		
Stage	OpenCV Average	Hiromoto
1	16	9
2	21	16
3	39	27
4	33	32
5	44	52
10	80	99
12	111	127
14	135	136

Table 4.9: FeaturespPer Stage Comparision

The work presented in [16] includes a table which shows the average number of clock cycles their hardware takes on average to process a single window. This normalizes the performance against the size of the input image and the scaling factor used. We too will calculate average clock cycles per window and then perform a final normalization to account for the clock-speed difference between the two designs, 125MHz for the present work and 160 MHz for [16]. The results of this are shown in Table 4.10 and Table 4.11.

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.

The performance of this work compared to [16] is very similar for the respective high

Our System Performance vs Cores, PixelRate = 4		
Cores	Cycles Per Window (125mHz)	Normalized to 160 mHz
4	11.93	15.27
8	7.94	10.17
12	6.41	8.20
16	5.42	6.94
20	5.03	6.44
24	4.48	5.73
28	4.08	5.22
32	3.81	4.88

Table 4.10: Cycles per Window in this Work

Hiromoto System Performance vs Number Of Parallel Stages	
Parallel stages	Cycles Per Window (160mHz)
1	339.1
2	200.3
3	119.6
4	69.5
5	41.3
6	26.0
7	16.2
8	10.4
9	6.9
10	4.7

Table 4.11: Cycles per Window In Hiromoto

end configurations. The relative cost of getting these performance levels will be evaluated, as well as how that cost scales with overall performance. Figure 4.9 shows the number of LUTs used as a function of the overall performance for both works. Similarly Figure 4.10 shows the number of registers used. In these figures points on the bottom-left represent high performance for low resource utilization.

Notice that that slope of the lines for both Figures for small numbers of cycles per window computation are similar, and very high. This means that the resource cost required to improve the performance of the object detection by a small amount is very high. Conversely however if an end user can accept slower performance the resource utilization savings for our system are far greater than in [16]. This is a result which is

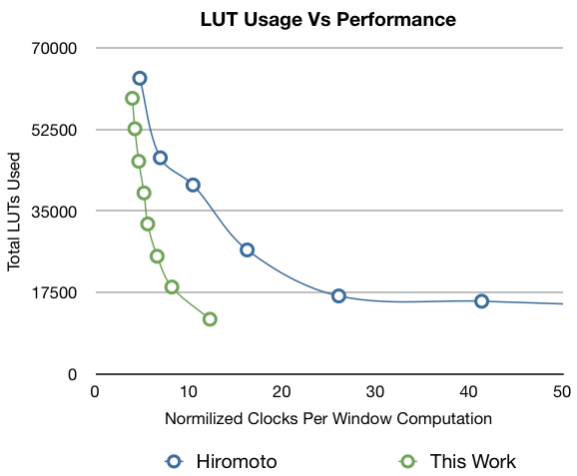


Figure 4.9: Register Utilization

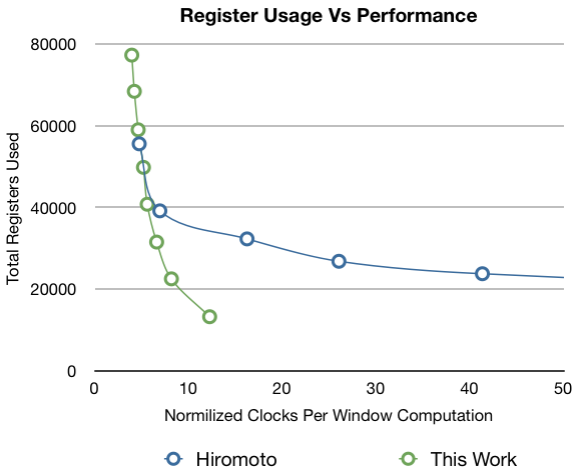


Figure 4.10: Lut Utilization

beneficial in the context of a mobile FPGA co-processor where resources are likely to be more limited.

Unfortunately the work done by Hiromoto [16] does not measure the power of their system, so a direct comparison cannot be done here. Qualitatively we might expect the power and energy usage of our system to be lower for two reasons. The first, as was just shown, is that the total resource utilization of our system tends to be better for a given performance level than in [16]. The second is that our system does not perform wasted calculations: each SIMD core will stop after the window it is working on has failed a stage. In [16] 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. For this reason the amount of raw computation in [16] and thus the dynamic power usage would likely be higher than in our work.

4.7 Summary

The experimental methodology and results of this work were presented in this Chapter. Our FPGA implementation of the OpenCV MSHCD algorithm was able to produce identical results while outperforming both a desktop computer by 3.8 times and mobile

phone by 59 times. Our system was able to outperform the mobile phone in the amount of energy it took to perform the MSHCD algorithm per image by 15 times, a key metric in the context of energy-constrained mobile devices. The high end configuration of our hardware system performed similarly to the most comparable previous work but due to our underlying architecture the resources required to implement our system scaled more favourably as less performance was required.

Chapter 5

Conclusions

The integration of multiple sensors with processors in programmable mobile devices has made smartphones increasingly important. The ability for a device to measure and understand the world around it has lead to a large number of novel and creative applications. Power constraints however, have limited the computational performance of these devices and in doing so has limited their usefulness in high bandwidth signal processing tasks, such as Computer Vision. This thesis proposes the inclusion of an FPGA co-processor in smartphones as a means of efficiently computing tasks such as Computer Vision. The MSHCD algorithm, as implemented in popular open source Computer Vision library OpenCV, was used as a large-scale real-world Computer Vision application.

5.1 Contributions

The key contributions of this work are

1. A novel architecture for performing the MSHCD algorithm on an FPGA. It exploited the parallelism in the MSHCD algorithm by loading neighbouring window caches in parallel and processing those window caches in a SIMD fashion. The result is a hardware system that outperforms a Nexus One smartphone by 59 times

while being 15 times more energy efficient.

2. A full object detection system that can use preexisting OpenCV compatible cascades and performs all of the steps computed by OpenCV. Steps such as image scaling and variance normalization calculations, which are not required in the core object detection algorithm, were implemented to ensure our system performed the same operations users have come to expect from software implementations.

3. Measurements of performance and energy of a large Computer Vision system across multiple computational media. These measurements motivate the usefulness of a potential FPGA co-processor in future mobile devices.

5.2 Future Work

In the future our hardware implementation of the MSHCD algorithm can be improved in two key ways: time-multiplexing memory, and dynamically generating integral images.

5.2.1 Time-Multiplexing Memory

Time-multiplexing memory refers to running a memory at twice the frequency of the logic that accesses it. It makes it possible for two independent sets of logic to read a memory from the same read port. Double pumping the integral image and classifier cascade would significantly increase the performance of our system by enabling two sets of SIMD PEs to access these memories independently. For example, two sets of 16 PEs (32 PEs total) could be processing the image and each set would have a relative performance, compared to a single core, of 13 times as was illustrated in Figure 4.3. Combined, the 32 total PEs would have a relative performance of 26 times, which is 45% faster than 32 PEs running as one SIMD block and uses the same amount of resources.

5.2.2 Dynamic Integral Image Generation

The integral and square integral images are the largest consumers of on-chip memory in the MSHCD hardware system. The access pattern of the integral image is well structured; the sliding window starts with windows at that top of the image and works its way down. While windows are sliding horizontally across a row of the integral image, all of the rows above the top of the window won't be accessed until the next scale iteration. When the sliding window moves down a row, the only additional data that it needs is one more row of the integral image.

The hardware system can exploit this by only instantiating enough memory to hold enough rows of the integral image such that the height fills the sliding window. The next row that the window needs can be dynamically generated while processing is done on the windows in the current row. The same principal can be applied to the square integral image and the total on-chip memory requirements of these two blocks can be significantly reduced, thus enabling the use of larger maximum images sizes.

Bibliography

[1] Qualcomm Snapdragon QSD8250 RISC Microprocessor with embedded DSP. pdadb.net/index.php?m=cpu&id=a8250&c=qualcomm_snapdragon_qsd8250, 2009.

[2] Cheng Auerbach, Bacon and Rabbah. Lime : a Java-Compatible and Synthesizable Language for Heterogeneous Architectures. *International Conference on Object Oriented Programming Systems Languages and Applications*, 45(10):89–108, 2010.

[3] Dan Bornstein. Brief overview of the Dalvik virtual machine and its insights. www.dalvikvm.com/, 2011.

[4] Isidor Buchman. The High-power Lithium-ion. batteryuniversity.com/learn/article/the_high_power_lithium_ion, 2011.

[5] Ed Burnette. Program for Android in C/C++ with the Native Development Kit. www.zdnet.com/blog/burnette/program-for-android-in-cc-with-the-native-development-kit-if-you-dare/1284, 2009.

[6] Chih-Rung Chen and Wei-Su Wong. A 0.64 mm 2 Real-Time Cascade Face Detection Design Based on Reduced Two-Field Extraction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(11):1937–1948, 2011.

[7] Chuan Cheng. An FPGA-based object detector with dynamic workload balancing. *Field-Programmable Technology (FPT)*, 2011.

[8] Peter Curry. Xilinx FPGA implementation of an image classifier for object detection applications. *International Conference on Image Processing*, pages 346–349, 2001.

[9] Google developer.android.com. What is Android. `developer.android.com/guide/basics/what-is-android.html`, 2012.

[10] Marc Ebner. Evolving Object Detectors with a GPU Accelerated Vision System. *Evolvable systems: from biology to hardware*, pages 109–120, 2010.

[11] Toronto Stock Exchange. Research In Motion Limited. `www.google.ca/finance?client=ob&q=TSE:RIM`, 2011.

[12] Fitzpatrick. An Interview with Steve Furber. *Communications of the ACM*, 54(5):34–39, 2011.

[13] Jeremy Ford. Android phone sells like hotcakes in kenya, the world next. `singularityhub.com/2011/08/16/80-android-phone-sells-like-hotcakes-in-kenya-the-world-next/`, 2011.

[14] Gall and Lempitsky. Class-specific Hough Forests for Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1022–1029, 2009.

[15] Willow Garage. OpenCV Wiki. `opencv.willowgarage.com/wiki/`, 2006.

[16] Masayuki Hiromoto. Partially Parallel Architecture for AdaBoost-Based Detection With Haar-Like Features. *IEEE Transactions on Circuits and Systems*, 19(1):41–52, 2009.

[17] Mathew Honan. Apple Unveils iPhone. `www.macworld.com/article/54769/2007/01/iphone.html`, 2007.

[18] Inteist. Android development vs. blackberry development. www.inteist.com/2009/04/android-development-vs-blackberry-development/, 2009.

[19] Rob Jackson. Android stealing symbian and winmo market share. phandroid.com/2009/11/15/android-stealing-symbian-winmo-market-share/, 2009.

[20] Sheng Liang. The Java Native Interface Programmer's Guide and Specification. www.zdnet.com/blog/burnette/program-for-android-in-cc-with-the-native-development-kit-if-you-dare/1284, 1999.

[21] Dylan Love. Android Has 47% Of The Mobile Market. articles.businessinsider.com/2011-12-30/tech/30571342_1_android-platform-smartphone-windows-phone, 2011.

[22] Mike Luttrell. 3.7 million android activations over christmas. www.tgdaily.com/mobility-brief/60454-37-million-android-activations-over-christmas, 2011.

[23] Subhransu Maji and Jitendra Malik. Object Detection using a Max-Margin Hough Transform. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1038–1045, 2009.

[24] Hugo Miller. RIMs BlackBerry Loses Further U.S. Market Share to Apple. www.bloomberg.com/news/2011-12-30/rim-s-u-s-market-share-falls-to-6-5-from-7-1-as-apple-samsung-advance.html, 2011.

[25] Andrew Munchbach. RIM and Apple top U.S. Smartphone market share. www.bgr.com/2009/10/28/rim-and-apple-top-u-s-smartphone-market-share/, 2011.

[26] Satnam Singh. Computing without Processors. http://insidehpc.com/2011/07/19/microsoft-accelerator-system-a-swiss-army-knife-for-heterogeneous-programming/, 2011.

[27] Patrick Sudowe. Efficient Use of Geometric Constraints for Sliding-Window Object Detection in Video. *International Conference on Computer Vision Systems*, pages 11–20, 2011.

[28] Takeuchi Yasue Kawahito Ishizaki Komatsu Suganuma, Ogasawara and Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[29] Terasic. Altera DE4 Development and Education Board. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=138&No=501&PartNo=2, 2011.

[30] Viola and Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2):137–154, 2001.

[31] Yu Wei. FPGA Implementation of AdaBoost for Detection of Face Biometrics. *IEEE International Workshop on Biomedical Circuits and Systems*, pages 6–9, 2004.

[32] Taylor Wimberly. JIT Performance Boost Coming With Android 2.2. androidandme.com/2010/05/news/jit-performance-boost-coming-with-android-2-2/, 2011.