Low-power high-resolution image detection

By

Caleb Merchant

A Thesis
Submitted to the Faculty of Mississippi State University in Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical Engineering and Computer Engineering in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2019
Low-power high-resolution
image detection

By
Caleb Merchant

Approved:

John E. Ball
(Major Professor)

Christopher Archibald
(Committee Member)

Ali Gurbuz
(Committee Member)

James E. Fowler
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering
Many image processing algorithms exist that can accurately detect humans and other objects such as vehicles and animals. Many of these algorithms require large amounts of processing often requiring hardware acceleration with powerful central processing units (CPUs), graphics processing units (GPUs), field programmable gate arrays (FPGAs), etc. Implementing an algorithm that can detect objects such as humans at longer ranges makes these hardware requirements even more strenuous as the numbers of pixels necessary to detect objects at both close ranges and long ranges is greatly increased. Comparing the performance of different low-power implementations can be used to determine a trade-off between performance and power. An image differencing algorithm is proposed along with selected low-power hardware that is capable of detected humans at ranges of 500 m. Multiple versions of the detection algorithm are implemented on the selected hardware and compared for run-time performance on a low-power system.
Key words: Object Detection, Image Detection, Low-Power, Long-Range, High-Resolution, Image Differencing, Frame Differencing, Morphology, Multi-Threading, Hardware Acceleration, ARM, Vivante, NXP, GPU, CPU
DEDICATION

To all of my professors at Mississippi State University who sparked my interest in Electrical and Computer Engineering and to my co-workers at Camgian for giving me a great work environment.
ACKNOWLEDGEMENTS

I would like to thank Camgian for allowing me to Co-Op as an undergraduate and work part-time for the past six years of my life as a high school student and a college undergraduate and graduate student. A lot of my motivation for pursuing Electrical and Computer Engineering came from the work environment at Camgian.

I would also like to thank Dr. John Ball who has helped me through this entire graduate school process. I have enjoyed having him as a professor at Mississippi State University, and his dedication to the Electrical and Computer Engineering Department is evident in the amount of work he puts in to everything.
CONTENTS

DEDICATION ................................................................. ii

ACKNOWLEDGEMENTS .................................................... iii

LIST OF TABLES ............................................................. vi

LIST OF FIGURES ........................................................ vii

CHAPTER

I. INTRODUCTION ......................................................... 1

II. BACKGROUND ........................................................... 4

   2.1 Image Differencing ................................................. 4
   2.2 Edge Detection ..................................................... 9
   2.3 Histogram of Oriented Gradients ................................. 10
   2.4 Image Differencing Thresholding ............................... 12
   2.5 Multi-Camera Processing ......................................... 14
   2.6 Convolutional Neural Networks ................................. 17
   2.7 Low-Power Mobile SOCs ......................................... 19
   2.8 Low-Power Hardware Accelerators .............................. 23

III. LOW POWER HIGH RESOLUTION IMAGE DETECTION IMPLEMENTATION .............................................. 24

   3.1 Introduction ....................................................... 24
   3.2 Proposed Solution ................................................ 25
      3.2.1 Hardware ...................................................... 25
      3.2.2 Software ...................................................... 28
      3.2.3 Tools and Framework ....................................... 31
         3.2.3.1 OpenCL .................................................. 32
         3.2.3.2 OpenCV ................................................ 32
      3.2.4 Implementation ............................................... 35
3.2.4.1 GPU with OpenCL ........................................ 35
3.2.4.2 CPU with OpenCV ....................................... 36
3.3 Results and Discussion ....................................... 37
  3.3.1 Analysis Criteria ......................................... 37
  3.3.2 Results .................................................... 38
3.4 Conclusions ................................................... 43

IV. CONCLUSIONS ................................................. 45
  4.1 Conclusions .................................................. 45
  4.2 Contributions ............................................... 46
  4.3 Future Work ................................................ 46

BIBLIOGRAPHY .................................................. 48
LIST OF TABLES

2.1 Mobile CPU specification evolution. ........................................ 20

3.1 CPU and GPU Run-time test case parameters. .......................... 38

3.2 CPU @ 1.0 GHz vs. GPU @ 800.0 MHz speedup factor. .......... 41

3.3 CPU @ 1.5 GHz vs. GPU @ 800.0 MHz speedup factor. .......... 41

3.4 OpenCL GPU @ 800.0 MHz run-time in milliseconds. ......... 41

3.5 OpenCV CPU @ 1.0 GHz run-time in milliseconds. ............ 42

3.6 OpenCV CPU @ 1.5 GHz run-time in milliseconds. ............ 42
LIST OF FIGURES

2.1 Image differencing processing chain. ........................................... 5

2.2 Background image differencing example where (a) is the background, (b) is
the image frame, (c) and (d) are the morphologically processed images, and
(e) shows the resulting bounding boxes. .......................................... 6

2.3 Frame image differencing example where (a) and (b) are image frames, (c) is
the morphed image, and (d) shows the resulting bounding boxes. ............ 7

2.4 Gaussian modeling image differencing example where (a) is the background,
(b) is the image frame, (c) and (d) are the morphed images, and (e) shows the
resulting bounding boxes. .............................................................. 8

2.5 HOG processing chain. .............................................................. 11

2.6 Smartphone CPU Performance. .................................................. 21

3.1 Image processing chain. .............................................................. 29

3.2 First image frame (left) and second image frame (right). .................... 32

3.3 Differenced image frames (top left), binary thresholded image (top right),
eroded image (bottom left), and dilated image (bottom right). .................. 33

3.4 Dilation image with AOI overlay. ................................................ 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Original image with AOI overlay.</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>Image processing chain threading model.</td>
<td>37</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

Detecting, localizing, tracking, and classifying objects in image and video streams is a topic that has thoroughly been researched by the technical community. Despite these efforts, research continues, because the solution is not complete. Many people even have these capabilities in their own homes in the form of security cameras. In many cases, these cameras must act autonomously in detecting objects without human assistance. Countless algorithms exist that can perform this task, and many of these algorithms are simple enough to run on battery-powered devices. The ability for a sensor to operate on battery power for extended periods of time is crucial in many applications for military and police forces, although other groups may find it desirable to have these capabilities.

Using an automated camera system that can detect objects at ranges similar to the human visual range (< 1000 m) can be useful as it removes the human requirement factor in surveillance and reconnaissance. To create a camera system with the ability to detect objects further than 100 m using only battery power, a system containing a high-resolution camera and powerful but low-power processing is ideal. With many of the state-of-the-art hardware accelerators, this problem becomes very solvable. Using higher-resolution cameras requires increased processing because of the increase in the total number of pixels.
One possible solution is to use an optical zoom instead of using a high-resolution camera; however, the optical zoom reduces the field of view (FOV) of the sensor. A camera with a wider FOV can scan a larger area for objects without the use of motors to adjust the camera’s FOV. The use of a motor or other similar mechanism would likely significantly reduce the battery life of the device and often can make the device less reliable.

Processing image frames from a high-resolution camera requires an increased amount of processing in contrast to lower-resolution cameras, but the battery life requirement limits the complexity that the processing can have. Therefore, an image processing methodology must be created that only requires limited processing, but still has the ability to perform object detection, localization, and possibly classification. Change detection between image frames is a trivial solution for solving this problem as it is simple but effective at finding objects of interest. The idea behind this approach is that a moving object in the FOV will change location, size, and/or intensity between two frames. This does not lend itself well to detecting stationary objects though. The assumption can be made for some cases that there are no objects in the FOV when the camera is on. Therefore, an object will have to move into the FOV to become a stationary object meaning that at some point, the object was identified. Another problem with this is approach is if the object is moving very slowly, it may not be detected if enough pixels have not changed between frames. Also, in order to reduce false alarms, some type of thresholding must be introduced that filters slower moving objects. Therefore, a balance between false positive and false negatives is crucial.

A solution to the problem is proposed based on frame image differencing. The difference between two frames captured during a short period away from each other are then
converted to a binary image using a dynamic threshold based on the mean and standard deviation of frame difference. Morphological operators are then used to reduce noise and amplify objects. Proposed object regions are then generated which would later be passed to a classifier.

My contributions include developing the methods to characterize each portion of the proposed solution on different platforms. From the characterization and evaluation of multiple software implementations, I devised an optimized solution to perform object detection on a low-power high-resolution camera using the i.MX 8 family of system on chips (SOCs) developed by NXP Semiconductors [13].
CHAPTER II

BACKGROUND

2.1 Image Differencing

Some of the most common image detection algorithms are based on the differencing method. The differencing method for image detection finds the objects of interest in an image based on the subtraction of two images. One method is to create a background image that new image frames are compared to where an image frame is defined as one of the still images captured by a camera. If a pixel or set of pixels has changed since the background was captured by some threshold, the pixel(s) are considered to be of interest. Further processing can be done such as morphological operations that can remove noise [4]. Without this morphological stage, slight changes in scenery would be considered objects of interest. Figure 2.1 shows a simplified processing chain for this image detection methodology [4].

Even though the morphology reduces some noise caused by a changing background, this method is still very susceptible to noise as in Figure 2.2, where it is shown that the sunlight reflecting off a window is considered an object of interest [4]. Another limitation of this method is that if the background changes over time, as it would with a visual camera with changing sunlight, a new background image must be captured. Capturing a new background image may be difficult because the camera must know that no objects of
interest are in its FOV. If an object of interest is in the FOV when the background image is captured, false alarms will occur when the object of interest moves out of the FOV. Therefore, human interaction would likely have to take place where a person is forced to specify the background image. While this approach is simple to implement and run on hardware, it is too susceptible to noise.

Another differencing method is to use the difference between two image frames. This method works the same way as the background image differencing except two frames are used instead of one frame and a background image. This method is less susceptible to noise but fails to segment objects perfectly [4]. In an application where only bounding boxes are required, this limitation can be ignored. Figure 2.3 shows an example of using frame-to-frame image differencing [4]. In contrast to background image differencing, it is shown...
Figure 2.2: Background image differencing example where (a) is the background, (b) is the image frame, (c) and (d) are the morphologically processed images, and (e) shows the resulting bounding boxes.
frame-to-frame image differencing is less susceptible to noise but fails to properly segment objects [4]. The proposed solution is based on this methodology as object segmentation is not of as much concern as detection. The proposed solution seeks to draw bounding boxes around objects of interest that can then be transmitted to an operator or classified by a further processing stage.

![Frame differencing example](image)

Figure 2.3: Frame image differencing example where (a) and (b) are image frames, (c) is the morphed image, and (d) shows the resulting bounding boxes.

One method of improving upon background image differencing is Gaussian background modeling. Instead of using a single background image, Gaussian background modeling is a statistical method that constantly updates the background image with the
Figure 2.4: Gaussian modeling image differencing example where (a) is the background, (b) is the image frame, (c) and (d) are the morphed images, and (e) shows the resulting bounding boxes.
changing environment [4]. Its processing chain is identical to that of the background image differencing except the background is constantly changing based on newly captured frames. Figure 2.4 shows an example of using Gaussian background modeling [4]. In this example, the segmentation is as good as background image differencing, and the low noise floor seen in frame-to-frame image differencing is also present. The proposed solution uses this idea of statistical modeling of the background to reduce noise.

Regional image differencing processing is used by Sengar [23] where statistics for local regions are used in processing which is similar to the proposed solution. However, Sengar [23] uses three-frame image differencing which uses the difference between the current frame and two previous frames. This helps reduce non-ideal effects from two frame differencing such as clutter and ghosting but requires more processing [23]. Many other solutions, such as the solutions described by Lin [11] and Abdelli [1], use this multi-frame image differencing approach to improve performance, but the proposed solution avoids multi-frame differencing to reduce processing requirements.

2.2 Edge Detection

Another common way to detect objects of interest is edge detection. Canny edge detection is one of the most common methods and still one of the most optimal [24]. Canny edge detection and other edge detection algorithms look for objects that do not “blend” with the environment and create edges. This method requires image gradients to be computed which generally requires more processing power than simple image differencing. Many of the first edge detections algorithms were susceptible to noise due to the gradient
operator. Examples include the Roberts Operator, the Prewitt Operator, and the Sobel Operator [24]. This is because they were all based on gradient operators which are sensitive to noise [24]. The Canny edge detector mitigates some of the effects of noise by applying a Gaussian filter to remove the noise before the edge detection occurs. While gradient-based edge detectors use first derivatives, Laplacian-based edge detectors use second order derivatives, though they also suffer from the same noise effects as gradient-based edge detectors [24]. Therefore, noise should be reduced before using these edge detectors. Many of these simple edge detectors can be kernelized and therefore calculated very efficiently [24]. Kernelization of an edge operator means that a matrix can be defined for an edge operator that can directly be convolved with an image. This two-dimensional sliding window convolutional operation is often an operation that is well optimized on computational hardware platforms, because one-dimensional and two-dimensional convolution maintains locality of reference and only consists of simple multiply accumulate operations. However, due to the long range image detection requirement, many of these simple edge transforms would not be viable as they would have difficulty separating an object of interest in the distance versus foliage blowing in the wind.

2.3 Histogram of Oriented Gradients

The Histogram of Oriented Gradients (HOG) method is an image detection processing scheme that has been used widely in recent years for its ability to detect humans, though it can be extended to detect and classify other types of objects using a classifier like a Support Vector Machine (SVM). HOG is possible to implement in many smaller, low-
power devices depending on the requirements. The idea behind the HOG method is that knowledge of local gradients is enough to perform object detection instead of computing gradients over each pixel in an image [3].

Figure 2.5 adapted from Dalal [3] shows the processing chain for implementing the HOG method. HOG divides an image into small spatial regions called “cells” and defines a local gradient for the entire cell based on the pixels within the cell [3]. Pre-processing must be done on the image which includes gamma and color normalization. Multiple “cells” are combined into larger “blocks” where each “cell” within the block is normalized based on the larger “block”. This process is repeated for overlapping windows in the image.

![HOG processing chain](image)

Figure 2.5: HOG processing chain.
HOG often outperforms wavelets and can be implemented somewhat easily which makes a useful tool for image detection [3]. HOG parameters can also improve run-time and accuracy performance in certain situations. For example, fine rather than coarse spatial sampling can improve human detection performance [3]. Also, “cell grid” overlapping and stride factor tweaking can further improve human detection performance [3]. Performance trade-offs in run-time and accuracy can help in determining an “operating point” to best utilize the HOG method for a specific application.

2.4 Image Differencing Thresholding

Many variations of the image differencing algorithms have been created. One example described by Luo [12] defines a gray threshold in which a global thresholding method is used. This reduces the noise effects and illuminates changes in the image subtraction. Equations 2.1, 2.2, 2.3, and 2.4 adapted from Luo [12] describe the global thresholding methodology where \( I(x, y) \) and \( J(x, y) \) are different image frames captured closely in time, and \( D(i, j) \) is the resulting detection image. The mean \( \mu \) and variance \( \sigma \) is calculated over the entire image given by width \( W \) and height \( H \). \( \alpha \) is a constant generally set to \( 1.1 \sim 1.2 \) that sets the sensitivity of the threshold function. \( P(i, j) \) is the difference between the frames \( I(x, y) \) and \( J(x, y) \).

\[
D(i, j) = \begin{cases} 
1 & |I(x, y) - J(x, y)| > T \\
0 & |I(x, y) - J(x, y)| \leq T 
\end{cases} \quad (2.1)
\]

\[
T = \frac{\mu + \sigma}{\alpha} \quad (2.2)
\]
\[
\mu = \frac{1}{W \times H} \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} P(i, j)
\] (2.3)

\[
\sigma = \frac{1}{W \times H} \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} |P(i, j) - \mu|
\] (2.4)

One limitation of using global mean and variance calculations is that the mean and variance could vary heavily over the entire image, especially if the image was captured with a wide-FOV camera. If half of the image has a much larger mean than the other half, the computed threshold would likely be too small for the larger mean portions but too large for the smaller mean portions. Therefore, dividing the image into multiple regions and calculating local means and variances could be very useful as it would mitigate some of the issues with a single global image threshold function. Equations 2.5, 2.6, 2.7, and 2.8 describe the resulting equations where \( h \) is the number of vertical and \( w \) is the number of horizontal blocks to divide the image. The \( \lfloor \rfloor \) operator is the floor function which is a truncation rounding. The morphological operations that are commonly used in image differencing-based processing chains are not described by Luo [12] meaning either they were excluded because the global threshold removed enough noise or was not discussed due to the focus on the thresholding method.

\[
D(i, j) = \begin{cases} 
1 & |I(x, y) - J(x, y)| > T(\lfloor x \times w/W \rfloor, \lfloor y \times h/H \rfloor) \\
0 & |I(x, y) - J(x, y)| \leq T(\lfloor x \times w/W \rfloor, \lfloor y \times h/H \rfloor)
\end{cases}
\] (2.5)

\[
T(a, b) = \frac{\mu(a, b) + \sigma(a, b)}{\alpha}
\] (2.6)
\[ \mu(a, b) = \frac{1}{W/w \times H/h} \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} P(a \times W/w + x, b \times H/h + y) \quad (2.7) \]

\[ \sigma(a, b) = \frac{1}{W/w \times H/h} \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} |P(a \times W/w + x, b \times H/h + y) - \mu(a, b)| \quad (2.8) \]

### 2.5 Multi-Camera Processing

Long-range object detection using visible and infrared images is difficult as the amount of processing necessary given the resolution requirements. It also becomes difficult to achieve a low false alarm rate. The Adaptive Double Structuring Element Top-Hat Transform (Adapt-Sel-DSTHT) is one way to tackle this problem for both infrared and visible images given the range of the object of interest is unknown [22]. Adapt-Sel-DSTHT takes the image frame and subtracts the image frame gray scale dilated then eroded by two different structuring elements. The structuring elements for the erosion and dilation are very important to the performance of this method and is what separates it from normal image differencing with morphological operators which requires the object of interest to have a known size for good performance [22]. In a highly cluttered environment Adapt-Sel-DSTHT can be modified to use a prediction of the background based on variance [22]. This comes at a cost of increased processing. Using the Adapt-Sel-DSTHT would be an effective long-range object detector as this method could possibly outperform the proposed solution in accuracy. However, the hardware for the proposed solution would likely not be powerful enough due to the “additional rule” and entropy steps [22]. While these operations are computationally simple, the proposed solution, which has a similar number
of operations as just the Adapt-Sel-DSTHT portion of the algorithm, already reaches the limits of real-time processing. Adding further algorithms that must traverse a large image would require more powerful hardware to be selected, but this would increase power consumption. Reducing the size of the input image to accommodate this algorithm would require either a reduction in range or reduction in FOV.

Adapt-Sel-DSTHT is defined by Equation 2.9 and Equation 2.10 adapted from Saran [22] where \( \bullet \) is the dilation operator, \( \circ \) is the erosion operator, \( f \) is the image, \( k \) is the frame number, and \( \lambda \) is the maximum object size.

\[
Adapt - Sel - DSTHT_k(x, y) = f_k(x, y) - (f \bullet \Delta B(x, y, \lambda)) \circ B_s(x, y, \lambda) \tag{2.9}
\]

\[
B_s(x, y, \lambda) = \begin{cases} 
1 & (x + 2i, y + 2j), \text{where } i, j \in [-\lfloor \frac{\lambda - 1}{4} \rfloor, \lfloor \frac{\lambda + 1}{4} \rfloor] \\
0 & \text{otherwise}
\end{cases} \tag{2.10}
\]

Further improvements in human and vehicle detection can be made with the use of visible and infrared camera combinations. Infrared cameras by themselves are prone to noisy, low quality images which makes it difficult to detect objects. However, humans and vehicles can be easily discerned in infrared cameras as they often are thermally different than their environment. Therefore, initial detection could be performed by the visible camera then confirmed by the infrared camera. This follows the idea of m-of-n processing where both cameras must produce a detection. M-of-n processing helps filter false alarms, clutter, and noise from both cameras.
The infrared image detection solution proposed by Zhang [26] uses frame differencing with an estimated background. The estimated background is computed using a morphological operator on the current image as the noise and object of interest in the image, given the object is small, can be removed leaving only the background [26]. The image is then differenced with the estimated background which will then result in an image showing the object of interest plus noise. The object of interest can then be separated from the noise using a sliding window to calculate local energy values. This assumes that the noise energy over a small region is smaller than the object of interest energy. This approach offers the benefit of being simple to implement on limited low-power hardware as the processing includes only morphology, differencing, and summation. It also attempts to find a way to separate the background, object of interest, and noise which is often the difficulty in using infrared images for detection. However, this solution is highly dependent on the morphological structuring elements utilized and the size of the object of interest in the image. If the object of interest is smaller in the image, the size of the window used by the sliding window summation would likely need to be smaller than if the object of interest was large in the image.

Some information about possible objects of interest is assumed in the proposed solution from other sensors or human interaction which makes this a viable method. The limiting part of this method is the creation of the background image. Using a static background image is not ideal for infrared images or even visual images because lighting and thermal levels can greatly change the background. Using a static background that is updated based on image variances is often inaccurate. Therefore, using a morphological operation to gen-
erate a background image is interesting, but there is not a single structuring element that is perfect for every situation. One situation may greatly benefit from a certain structuring element while another situation may see poor performance with that same structuring element. If the background is not generated well, the separation of the noise and object of interest becomes impossible as the assumption that noise has less local energy than an object may not hold. However, if the background can properly be modeled, this solution can be very useful as many other methods fail to detect objects of interest in infrared images because of noise sensitivity.

2.6 Convolutional Neural Networks

Traditional object detection and classification relies on extracting features in an image. These features are often hard to determine, and a lot of work is required to gather and evaluate different features. Deep learning in the form of convolutional neural networks overcomes this difficulty by training a pre-defined network to learn the important features in images. Convolutional neural network design is loosely inspired by the mammalian visual system [5].

Three main layers exists in most modern convolutional neural networks: convolutional layers, pooling layers, and fully-connected layers [5]. The goal of the convolutional layers is to extract features from an image. A large number of features are produced, and many do not provide much insight into object detection and classification. These features are filtering by pooling layers which reduce the dimensionality of the features. Finally, the fully-connected layers take these features and form a classifier often using multi-layer
perceptrons concepts. Gui [5] provides a simple implementation of a convolutional neural network. More advanced convolutional neural networks such as YOLOv3 [20], provide some of the most accurate object detection and classification results. YOLOv3 extends the capabilities of the original YOLO architecture [18] and YOLO9000 [19].

YOLOv3 claims mAP at 0.5 IOU of 57.9 with an inference time of 51 ms for images of size $608 \times 608$ [20]. IOU is defined as the intersection over union which is the overlap between a predicted bounding box and the ground truth bounding box. mAP is defined as mean average precision of classification. Both of thesis metrics are described by Tan [25]. While the YOLOv3 network inference time shows real-time performance capabilities, these numbers were gathered from a machine with a high-performance GPU. Running this network on a low-power CPU or GPU would likely take multiple seconds. TinyYOLOv3 is another available network architecture that is stripped down version of YOLOv3 with less accuracy that can be ran on a low-power CPU or GPU with real-time performance [20]. However, tinyYOLOv3 generally requires images less than one megapixel. The proposed solution requires images that have tens of megapixels to detect objects at long-range. Using larger images with tinyYOLOv3 would increase the run-time and reduce the real-time capabilities. While an accurate convolutional network cannot run in real-time on a low-power CPU or GPU given an image of tens of megapixels, an object can be detected and then classified in a smaller image that is pre-processed and cropped using a method such as frame differencing. Instead of the entire image being sent to the network for detection and classification, only the portions with objects of interest detected
by frame differencing. If a convolutional neural network is used for localization and classification, this would mitigate the localization issues with frame-to-frame differencing.

Further improvements upon YOLOv3 have also been made for specific situations. Qu defines a method for detecting humans using image enhancement by Retinex [17]. The image enhancement reduces the false alarm rate, but the YOLO network still has difficulty detecting objects that are close together [17]. The proposed solution does not solve this problem either as frame-to-frame differencing makes it difficult to determine exact object location. Kim also presented modifications to YOLOv3 by adding predication layers for vehicle detection [10]. This improves the performance of the network architecture, but also makes it less feasible for computationally limited platforms because of the added layers.

2.7 Low-Power Mobile SOCs

The goal of any low-power embedded CPU is to fit more capabilities in a smaller form factor while using as little power as possible. Mobile phone CPUs is one of the major markets in which low-power embedded processors are important. Table 2.1 adapted from Halpern [6] shows the evolution of mobile CPU specs. From this, it is shown that mobile CPU performance and capabilities have greatly increased in the past decade.

Recent improvements of mobile CPUs have slowed down in recent years. Many mobile SOCs have reached a “power wall” at 1.5 W of dynamic power which appears to be the best balance between power consumption and capabilities [6]. To improve the performance of mobile CPUs, many vendors have adopted desktop CPU optimization techniques such as
advanced pipelining and out of order instruction execution. Figure 2.6 shows the speedup seen by different generations of ARM Cortex processors in smartphones [6].

Many mobile processors use frequency scaling to reduce power consumption. Frequency scaling alone does not reduce dynamic power consumption, because the frequency and execution time are tightly coupled. Frequency scaling does allow for voltage scaling which can reduce power consumption. From Equation 2.11 where \( P_D \) is the dynamic power, \( C_{PD} \) is the total power dissipation capacitance, \( V_S \) is the supply voltage, \( f_C \) is the clock frequency, and \( \alpha \) is the activity factor, we can see that the square of the voltage influences the power consumption of a mobile CPU. Therefore, small reductions in voltage can greatly reduce power consumption. When reducing the voltage to a mobile CPU, the maximum frequency at which the CPU operates decreases. This is why frequency scaling must be introduced to perform voltage scaling which can then reduce power consumption.

\[
P_D = C_{PD} \times V_S^2 \times f_C \times \alpha
\]  

(2.11)
Mobile CPUs have very strict thermal design constraints. This is one of the main reasons for the 1.5 W “power wall” that has been reached in recent years [6]. However, many mobile CPUs have the capability to exceed this constraint with CPU frequencies reaching that of many desktop CPUs. The assumption is that averaged over time, the CPU will not exceed 1.5 W, and this assumption is also enforced by many mobile CPU operating system configurations. Allowing the CPU to occasionally exceed the 1.5 W power consumption barrier means that time critical application can be executed at accelerated clock frequencies as long as they are infrequent. This concept is important in the proposed solution. If an object is detected in an image somewhat infrequently, more advanced techniques can be used to classify objects than were used to detect objects such as convolutional neural networks.
While the goal of many smartphone processors is to increase the user experience by adding more capabilities in a battery powered environment, energy efficient high-performance computing (HPC) has been of importance in recent years. ARM has been the dominating core architecture of many low-power devices but has also gained some traction in the HPC realm with the introduction of ARM based supercomputers [8]. One common benchmark used for scientific workloads on supercomputers is Gaussian Elimination and is used by Johnston [8] to compare multiple low-power CPUs and GPUs.

It is often expected that GPUs will generally outperform CPUs at computational-based tasks, even in the low-power realm. Based on the results by Johnston [8], most low-power GPUs do not offer much benefit over low-power CPUs in both execution time and power efficiency. The lack of GPU performance boosts in the low-power realm was also discussed by Halpern [6] where it is stated that GPU performance does not increase the user’s experience as much as CPU performance, even in graphics-based applications. Johnston [8] compares Vivante and Mali GPUs with ARMv7 cores, which are two of the most common GPU vendors found in ARM SOCs. There still exists a benefit for having low-power GPUs on SOCs in the form of parallel operation with CPU. While low-power CPU cores and GPU cores have similar performance and power efficiency, GPUs can be used in parallel with CPUs to offload computational tasks instead of bottle-necking the CPUs performing the tasks.
2.8 Low-Power Hardware Accelerators

Other low-power hardware accelerators exist, but most of these surpass the 1.5 W “power wall” that has been set by mobile phone companies. The Nvidia Jetson TX1 and TX2 are two examples of powerful, but low-power (7 W to 15 W) hardware accelerators. Rungsuptawee-koon [21] performed an evaluation of these two platforms in the realm of deep learning. The Nvidia Jetson TX1 and TX2 use the Nvidia Tegra X1 and X2 SOCs respectively [21]. The Nvidia Tegra SOCs give convolutional neural network inference capabilities to low-power devices, as long as they are capable of breaking the 1.5 W “power wall”. They offer high power efficiency, while maintaining high performance, though they do not offer the same capabilities as many desktop class GPUs.

The Intel Movidius Myriad processors are other hardware accelerators used for neural network inference. Both processors boast less than 1 W performance while still being able to perform neural network inference in real-time. The Intel Movidius Myriad processors were designed with computer vision in mind, but with their SIMD architecture and parallel instruction issuing capabilities, they can be used for other scientific computing tasks [7]. The first-generation Intel Movidius Myriad processor has comparable energy efficiency to the Nvidia Tegra processors [7]. However, very little published work on the newest Intel Movidius Myriad processor has been done which boasts much greater performance and efficiency compared to the first-generation architecture.
CHAPTER III

LOW POWER HIGH RESOLUTION IMAGE DETECTION IMPLEMENTATION

3.1 Introduction

Low-power image and video detection is often desired for security and remote surveillance as it offers a way to detect threats and possibly classify them while having hardware that is battery powered and is on for extended periods of time, meaning that it can operate in remote environments. If human detection can be performed on battery power alone, then systems can be deployed into remote locations that have the ability to autonomously detect humans or integrate with sensor networks to provide a detailed picture of an environment.

To perform longer range human detection (> 100 m), high-resolution cameras and/or cameras with large optical zooms are required. However, large optical zooms are not ideal for situations that require wide-FOV detection, because the optical zoom focuses the camera to a certain location. Optical zoom also requires the use of motors which reduces reliability and increases power consumption. Long-range wide-FOV image detection requires a high-resolution camera with a large enough FOV for the specified application. Given that the camera must have high-resolution, the amount of processing power required to detect objects becomes a lot greater. Increasing the processing requirements requires more processing power which reduces the ability for the camera to operate on battery power for an extended period of time. Therefore, a proposed solution to the problem must wield a
tight balance between processing power and battery life that can only be achieved with the state-of-the-art CPUs, GPUs, FPGAs, and other hardware accelerators.

While many state-of-the-art low-power hardware accelerators exist, long battery life will have to be maintained by having a simple image processing chain that has limited processing and power requirements. One obvious solution that has gained a lot of traction in recent years is convolutional neural networks. However, running a convolutional neural network on high-resolution images would be nearly impossible for a battery powered device. Many convolution networks that are ran low-power devices use images less than 1 MP. A long-range wide-FOV capable camera will likely need to have a resolution of > 10 MP.

3.2 Proposed Solution

Hardware selection was done based on research on power consumption and SOC capabilities. Many companies, such as Texas Instruments and NXP Semiconductors, invest a lot of resources in characterizing the power versus capabilities profile of chips. This previous research was leveraged in the hardware selection. The software libraries chosen for the proposed solution have been strenuously tested by the community and have been well optimized for computational tasks.

3.2.1 Hardware

The proposed solution utilizes an NXP i.MX 8M Quad SOC with a high-resolution (20 MP) visual camera and an infrared camera. The NXP i.MX 8M Quad SOC contains four ARM Cortex-A53s and a Vivante GC7000L GPU [13]. The ARM Cortex-A53s are 64-bit
processors based on the ARMv8-A architecture. The NXP i.MX 8M Quad SOC boasts a power consumption of 1.5 W for the four-core Dhrystone benchmark including DDR memory power consumption (1.0 W for the SOC) [14]. Running the four-core Dhrystone benchmark alongside a GPU benchmark boasts a power consumption of 3.4 W including DDR memory power consumption [14]. This makes the SOC ideal for low-power processing and is why it was selected for the proposed solution. Both the CPU core complex and the GPU will be evaluated in the proposed solution.

The NXP i.MX 8M has similar specs to many of the new Raspberry Pi boards as well as mobile phones which both often use 64-bit ARMv8 based architectures. Therefore, many of the results produced by the proposed solution could easily be translated to many of the common low-power platforms. The Raspberry Pi boards and many mobile phones often utilize different GPUs though. Therefore, the GPU results may not be as translatable to other low-power platforms, but the Vivante GPU analysis may still be useful for comparing the performance of other GPUs.

Each ARM Cortex-A53 contains an ARM Neon co-processor with a SIMD architecture [2]. This makes it useful for image processing tasks that can be parallelized. The theoretical maximum operations per second can be computed using Equation 3.1 where $F_c$ is the clock frequency, $n$ is the number of operations that can be performed in parallel, and $p$ is the number of processors. Memory widths, cache performance, and pipeline latency can limit the throughput of the Neon co-processor. Also, some Neon architectures allow multiple instructions to be issued in a single clock cycle which can improve the theoretical throughput.
The i.MX 8M is contains four Neon co-processors operating at a maximum frequency of 1.5 GHz [13]. The Neon architecture uses 64-bit or 128-bit wide registers [2]. Therefore, if 8-bit data is being used, up to 16 8-bit values can fit into a single register. The Neon architecture supports 8 or 16 simultaneous operations on a register of 8-bit values. Therefore, when using 8-bit unsigned data on the i.MX 8M, \( n = 16 \), \( p = 4 \), and \( F_c = 1.5 \times 10^9 \text{Hz} \) for Equation 3.1 which gives a throughput of 96 billion operations per second. For 32-bit floating-point data, \( n = 4 \) which gives a throughput of 24 billion operations per second using Equation 3.1.

The i.MX 8M contains a single Vivante GC7000L GPU [13]. The maximum throughput of GC7000L GPU can also be calculated using Equation 3.1 by setting \( F_c = 1.0 \times 10^9 \text{Hz} \), \( p = 16 \), and \( n = 4 \) for 32-bit floating-point values and \( n = 1 \) for 8-bit integer operations. This gives a maximum theoretical throughput of 64 billion 32-bit floating-point and 16 billion 8-bit integer operations per second. However, the i.MX 8M Vivante Linux Kernel module limits the GC7000L clock frequency to 800 MHz which reduces the theoretical throughput by 20 percent. This was likely selected to save power on the i.MX 8M and reduce the heat sink requirements. A custom board could use the full 1 GHz clock, but 800 MHz was used for the proposed solution. The GC7000L contains 16 Vega shader cores that can perform four simultaneous floating-point operations but only supports a single integer operation per core per clock cycle [15]. Therefore, it is beneficial to utilize floating-point math when using the GPU.
Given these theoretical performances, the i.MX 8M CPU core complex should have better throughput for images represented as an array of unsigned 8-bit integers, but the i.MX 8M GPU complex should have better performance for images represented as an array of normalized 32-bit floating-point values.

3.2.2 Software

The image processing chain consist of six steps as shown in Figure 3.1: (1) image differencing, (2) local mean and standard deviation calculations, (3) global and/or local threshold calculating, (4) image thresholding, (5) morphological opening, and (6) detection counting. Equations 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7 give a mathematical representation of the image processing chain where \( f_k \) is an image frame, \( H \) and \( W \) are the image dimensions, \( h \) and \( w \) are the local statistical region size dimensions, \( \lfloor \cdot \rfloor \) is the floor operation, \( \circ \) is the erosion operator, \( \bullet \) is the dilation operator, and \( S_E \) and \( S_D \) are the erosion and dilation kernels respectively. Zero padding can be used if \( h \) and \( w \) are not integer multiples of \( H \) and \( W \) respectively. \( T(\mu_{a,b}, \sigma_{a,b}) \) is the global or local image difference threshold function, \( CT(\mu, \sigma, w, h) \) is the function that defines the number of pixels that must be set in a local region to be considered an area of interest (AOI), and \( Det(a, b) \) maps the resulting AOIs. An AOI is a region in an image that likely has an object of interest present. AOIs can be clustered together then processed by further processing steps that can localize and classify objects. All of these steps can greatly be improved by parallel processing except for step three, although step three can benefit some from parallel processing. The proposed solution is based on frame-to-frame image differencing with background modeling thresholds.
which is discussed by Gau [4] and Luo [12]. The statistical region size will be application
dependent and could vary with the environment.

\[
D(i, j) = \left| f_k(i, j) - f_{k-1}(i, j) \right| \quad (3.2)
\]

\[
\mu(a, b) = \frac{1}{W/w \times H/h} \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} D(a \times H/h + y, b \times W/w + x) \quad (3.3)
\]

\[
\sigma(a, b) = \frac{1}{W/w \times H/h} \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} \left| D(a \times H/h + y, b \times W/w + x) - \mu(a, b) \right| \quad (3.4)
\]

\[
B(i, j) = \begin{cases} 
1 & D(i, j) \geq T(\mu([i \times h/H], [j \times w/W]), \sigma([i \times h/H], [j \times w/W])) \\
0 & D(i, j) < T(\mu([i \times h/H], [j \times w/W]), \sigma([i \times h/H], [j \times w/W]))
\end{cases} 
\quad (3.5)
\]
\[ M(i, j) = (B(i, j) \circ S_E) \cdot S_D \quad (3.6) \]

\[
\begin{align*}
Det(a, b) = & \begin{cases} 
1 & \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} M(y, x) \geq CT(\mu, \sigma, w, h) \\
0 & \sum_{x=0}^{W/w-1} \sum_{y=0}^{H/h-1} M(y, x) < CT(\mu, \sigma, w, h)
\end{cases}
\end{align*} \quad (3.7)
\]

The image differencing step takes the absolute difference between two frames in a video stream. The time difference in frames can vary depending on the desired motion tracking. A set of frame time differences will be used in the final version of the image processing chain to capture objects moving at different speeds, and the time differences may change based upon the environment. The next step computes local statistics for local regions. Each region consists of rectangular subset of pixels in the image, but can modified to other shapes for different applications. From the mean and standard deviation of each local region, a local or global threshold is calculated. The threshold(s) are then compared against the entire image difference result. If the difference between the two images is greater than the threshold, then the corresponding pixel is set to one and otherwise set to zero. Next, the morphological open operation removes noise from the image and amplifies differences. An open operation consists of first an erosion then a dilation step. Areas where only a few pixels are different will be removed by the erosion, and areas with large differences will be amplified by the dilation. A square kernel was used for both the erosion and dilation steps. The size of the kernel is given by the erosion and dilation factors where the kernel matrix dimensions is given by \( \text{dim}(\text{Kernel}) = (2 \cdot \text{factor} + 1) \times (2 \cdot \text{factor} + 1) \). The number of pixels set to one is then counted in each local region from the morphologi-
cally opened image. Local regions that have more than some percentage of pixels, say 30 percent, that are set to one are considered to be AOIs. Adjacent AOIs can be clustered into larger regions. This clustering algorithm will likely be application dependent.

Figure 3.2 shows an example of two frames that could be processing by the proposed solution, Figure 3.3 shows the image representation of the result at each step, Figure 3.4 shows the dilated image with the AOIs overlaid in gray, and Figure 3.5 shows the original frame with the AOIs overlaid in red. From these figures, obvious limitations of the approach can be seen. When using the absolute difference between two image frames, if an object of interest has moved significantly between frames, the AOI becomes much larger. One solution is to ensure the object motion is limited between frames by using a small time delta between frames. This is why using multiple time differences between frames can be useful as the motion of a fast-moving object can be limited, but other deltas can detect slow moving objects. It is also important to note that in general, the dilation factor should be larger than the erosion factor to ensure the AOI is not too small relative to the object of interest size in the original image, but making the dilation factor too large can make the AOI too large.

3.2.3 Tools and Framework

Multiple GPU and CPU implementations of the image processing chain were evaluated. OpenCL [9] was used for GPU programming and OpenCV [16] was used for CPU programming. However, OpenCV also has the ability to use OpenCL as its algorithmic backend in some instances, and OpenCL can be used for CPU programming as well.
3.2.3.1 OpenCL

OpenCL was used to create the GPU enabled image processing chain. OpenCL is a framework for writing programs that utilize hardware accelerators and is managed by the Khronos Group [9]. OpenCL allows the programmer to select the platform each algorithm is ran on (i.e. CPU or GPU), but it requires the user to create the algorithms by hand. Functions that are executed on a hardware accelerator are called “kernels”. The programmer must write and compile each “kernel” and manage memory for each device. For the i.MX 8M board, OpenCL can utilize either the CPUs or GPU.

3.2.3.2 OpenCV

OpenCV was used to program the ARM CPUs with Neon co-processors, although OpenCV can be configured to use OpenCL under the hood which would allow for GPU usage. OpenCV is a library of functions that are aimed at computer vision meaning that
Figure 3.3: Differenced image frames (top left), binary thresholded image (top right), eroded image (bottom left), and dilated image (bottom right).
Figure 3.4: Dilation image with AOI overlay.

Figure 3.5: Original image with AOI overlay.
many common image processing algorithms are already included in the library. It was originally developed by Intel but has since been ported to many other platforms [16]. On the i.MX 8M, OpenCV utilizes the Neon co-processor on the ARM CPU to do vectorized mathematical operations using SIMD instructions.

### 3.2.4 Implementation

Multiple implementations using OpenCL and OpenCV were created. Initial proof of concept work was done to determine the best way to implement each algorithm.

#### 3.2.4.1 GPU with OpenCL

Three versions of the image processing chain were created in OpenCL for the GPU: an 8-bit unsigned integer version, a 32-bit floating-point version, and a four element 32-bit floating-point vector version. The Vivante GPUs support four element floating-point SIMD instructions, but only support single-instruction single-data (SISD) instructions for integers [3]. The Vivante GPU guide also mentions a few other optimizations that were utilized: (1) using native floating-point instructions instead of generic floating-point instructions, (2) using OpenCL buffers instead of OpenCL images, (3) using 16 byte read/writes to reduce read/write request, and (4) avoiding use of OpenCL local memory [15]. The GC7000L has a theoretical 32-bit floating-point performance of 64 billion operations per second and a theoretical integer performance of 16 billion operations per second. Therefore, floating-point operations should be used when possible, but using 32-bit floating-point values instead of 8-bit integer values increased memory usage which can also decrease performance.
3.2.4.2 CPU with OpenCV

A single OpenCV implementation was created that could use a variable amount of threads. Each processing step utilized unsigned 8-bit integers, except for the mean and standard deviation calculations which used 32-bit floating-point values, to limit the memory usage. Also, each step was optimized to maintain cache coherency. The following OpenCV functions were utilized: \texttt{absDiff()}, \texttt{meanStdDev()}, \texttt{compare()}, \texttt{erode()}, \texttt{dilate()}, and \texttt{countNonZero()}. The \texttt{absDiff()} function performs Equation 3.2, the \texttt{meanStdDev()} function performs Equation 3.3 and Equation 3.4, the \texttt{compare()} function performs Equation 3.5, \texttt{erode()} and \texttt{dilate()} performs Equation 3.6, and \texttt{countNonZero()} performs Equation 3.7. The \texttt{absDiff()} and \texttt{compare()} operations were performed on subsections of the image in parallel where each thread was given a subsection to process. For cache coherency, the image was divided by rows only and not columns. This is because the image was stored in row major order, and dividing by rows maintains locality of reference. The \texttt{meanStdDev()} and \texttt{countNonZero()} operations were performed on each processing region in the image. Each thread was given a subset of the regions to process. The best performance was seen when the local regions were separated by columns and not rows, but this would be application dependent based on how regions were divided. The \texttt{erode()} and \texttt{dilate()} operations are parallelizable with OpenCV, but initial tests did not show much performance gains from doing so. Therefore, the \texttt{erode()} and \texttt{dilate()} operations remained single threaded.

Figure 3.6 shows the threading model of the OpenCV implementation of the image processing chain. Note that after each processing step, a synchronization step occurs. It
is possible to combine the `absDiff()` and `meanStdDev()` steps without the use of synchronization by diving the image by local regions. However, this does not maintain locality of reference as well which made this implementation slower than the thread synchronization between the steps. Note that the `calcThreshold()` function generates the global and/or local thresholds used in the `compare()` function and will be application specific.

![Image processing chain threading model.](image)

**Figure 3.6**: Image processing chain threading model.

### 3.3 Results and Discussion

The run-time performance of both the CPU and GPU implementations were analyzed. The numerical accuracy of these two solutions should be the same, assuming they use the same data types and both comply to the IEEE floating-point standards; therefore, numerical accuracy measurements were not performed.

#### 3.3.1 Analysis Criteria

For the run-time tests, multiple cases exist where the block size and the dilation/erosion factors were varied. Table 3.1 shows the parameters for the “Case 1” and “Case 2” test cases for the GPU and CPU. The columns labeled “Case 1” show the best-case selected parameters, and the columns labeled “Case 2” show the worst-case selected parameters.
analyzed for run-time performance for the image processing chain implemented using the CPU and GPU. In general, for the CPU, larger block sizes are preferred to maintain cache coherency, but for the GPU, smaller block sizes are preferred to allow for more parallelism. The erode a dilate factors for both the CPU and GPU will likely also affect the run-time as the number of operations per pixel is given by Equation 3.8 where \( h \) is the image height, \( w \) is the image width, and \( f \) is the erosion/dilation factor. Note that the camera used to evaluate the algorithms has a resolution of \( 3672 \times 5496 \) pixels.

\[
Ops = hw(2f + 1)^2
\]  

(3.8)

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Erode Factor</th>
<th>Dilate Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Case 1</td>
<td>Case 1</td>
</tr>
<tr>
<td>CPU</td>
<td>153x229</td>
<td>24x12</td>
</tr>
<tr>
<td>GPU</td>
<td>24x12</td>
<td>153x229</td>
</tr>
</tbody>
</table>

### 3.3.2 Results

The results in Table 3.2 and Table 3.3 show that the CPU with the Neon co-processor almost always outperforms the GPU. This was to be expected as the CPU integer performance is greater than the GPUs. However, for operations for that require floating-point math like the mean and standard deviation calculations, the GPU often outperformed the CPU. For the CPU, algorithms with longer run-times greatly improved from paralleliza-
tion, but for algorithms with short run-times, parallelization did not add much improvement. This is to be expected as there is some overhead in creating or signaling threads to run, and parallelization also puts more strain on memory bandwidth and has scheduling overhead among other limitations. For the GPU, most of the operations benefited heavily from the utilizing the floating-point SIMD instructions, though the OpenCL compiler used SIMD instructions when it knew it could (i.e. absolute difference operation). It was also initially thought that copying memory between the GPU and CPU would be too slow. However, as long as no full-size images are copied from the GPU to the CPU, the memory copy time is negligible as the copy from CPU to GPU can be done in parallel to the previous processing cycle. The memory management likely could be optimized further in the OpenCL implementation by removing the image copy operations, because the CPU and GPU utilize shared memory on the i.MX 8 family. However, memory copy times were not considered in the run-time performance, so this optimization was not pursued.

The values in the Table 3.2 and Table 3.3 are computed by divided the CPU run-time by the best GPU run-time given in Equation 3.9 where $T_{CPU,<\text{u8}>}$ is the unsigned 8-bit integer CPU run-time, $T_{GPU,<\text{u8}>}$ is the unsigned 8-bit integer GPU run-time, $T_{GPU,<\text{f32}>}$ is the 32-bit floating-point GPU run-time, and $T_{GPU,<\text{f32}[4]>}$ is the four element 32-bit vector GPU run-time. Therefore, numbers larger than 1.0 denote the CPU outperformed the GPU, and numbers smaller than 1.0 denote the GPU outperformed the CPU. Three different data types were evaluated for the GPU: unsigned 8-bit integers (uint8), 32-bit floating-points (float32), and 32-bit floating-point vectors (float32[4]). For OpenCL, the “Read/Write” columns corresponding to the summed read and write times of the image
data and detection results respectively. The write operations took most of this time and the read operations only took 1-2 milliseconds. These numbers could be ignored if the OpenCL profile allowed for shared CPU and GPU memory, but this optimization was not explored as these run-times were negligible to the total run-time. For both OpenCV and OpenCL, the “AbsDiff” column corresponds to the image differencing step, the “Mean Std-dev” column corresponds to the mean and standard deviation step, the “Binary” column corresponds to the binary thresholding step, the “Erode” and “Dilate” columns correspond to the morphological operation steps, and the “Count” column corresponds to the detection counting step. The “Overall” column is the total run-time of the entire processing chain. Run-times were compared for one, two, three, and four CPU cores with the best-case performance parameters denoted by “Case 1” and worst-case performance parameters denoted by “Case 2”. The run-times of each implementation are given in milliseconds in Table 3.4, Table 3.5 and Table 3.6.

\[
S = \frac{T_{CPU<\text{u8}n}}{\min(T_{GPU<\text{u8}n}, T_{GPU<f32n}, T_{GPU<f32[4]n})} \quad (3.9)
\]

Given these results, the i.MX 8M Quad CPU core complex will be utilized for the image processing chain instead of the GPU. If the i.MX 8 QuadMax was utilized instead of the i.MX 8M Quad, the GPU performance would be increased a factor of four and the CPU performance would increase by 50 percent. The GPU performance could also be increased by another 25 percent by increasing the clock speed from 800 MHz to 1 GHz. Even with the increased GPU performance, the CPU cores would likely give better performance.
Table 3.2: CPU @ 1.0 GHz vs. GPU @ 800.0 MHz speedup factor.

<table>
<thead>
<tr>
<th>CPU Cores</th>
<th>Test Case</th>
<th>AbsDiff</th>
<th>Mean Stddev</th>
<th>Binary</th>
<th>Erode</th>
<th>Dilate</th>
<th>Count</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Case 1</td>
<td>10.00</td>
<td>0.43</td>
<td>8.92</td>
<td>7.77</td>
<td>7.77</td>
<td>2.83</td>
<td>4.71</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>10.00</td>
<td>0.11</td>
<td>8.92</td>
<td>11.59</td>
<td>17.32</td>
<td>0.29</td>
<td>5.63</td>
</tr>
<tr>
<td>2</td>
<td>Case 1</td>
<td>14.55</td>
<td>0.83</td>
<td>10.55</td>
<td>7.77</td>
<td>7.77</td>
<td>5.31</td>
<td>6.32</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>14.55</td>
<td>0.26</td>
<td>10.55</td>
<td>11.59</td>
<td>17.32</td>
<td>0.58</td>
<td>8.72</td>
</tr>
<tr>
<td>3</td>
<td>Case 1</td>
<td>14.55</td>
<td>1.23</td>
<td>12.89</td>
<td>7.77</td>
<td>7.77</td>
<td>7.73</td>
<td>7.17</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>14.55</td>
<td>0.39</td>
<td>12.89</td>
<td>11.59</td>
<td>17.32</td>
<td>0.86</td>
<td>10.30</td>
</tr>
<tr>
<td>4</td>
<td>Case 1</td>
<td>16.00</td>
<td>1.59</td>
<td>14.50</td>
<td>7.77</td>
<td>7.77</td>
<td>9.44</td>
<td>7.68</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>16.00</td>
<td>0.53</td>
<td>14.50</td>
<td>11.59</td>
<td>17.32</td>
<td>1.15</td>
<td>11.35</td>
</tr>
</tbody>
</table>

Table 3.3: CPU @ 1.5 GHz vs. GPU @ 800.0 MHz speedup factor.

<table>
<thead>
<tr>
<th>CPU Cores</th>
<th>Test Case</th>
<th>AbsDiff</th>
<th>Mean Stddev</th>
<th>Binary</th>
<th>Erode</th>
<th>Dilate</th>
<th>Count</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Case 1</td>
<td>11.43</td>
<td>0.62</td>
<td>12.89</td>
<td>10.80</td>
<td>10.80</td>
<td>4.05</td>
<td>6.62</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>11.43</td>
<td>0.19</td>
<td>12.89</td>
<td>16.83</td>
<td>25.49</td>
<td>0.41</td>
<td>8.73</td>
</tr>
<tr>
<td>2</td>
<td>Case 1</td>
<td>14.55</td>
<td>1.23</td>
<td>14.50</td>
<td>10.80</td>
<td>10.80</td>
<td>7.08</td>
<td>8.72</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>14.55</td>
<td>0.38</td>
<td>14.50</td>
<td>16.83</td>
<td>25.49</td>
<td>0.81</td>
<td>12.62</td>
</tr>
<tr>
<td>3</td>
<td>Case 1</td>
<td>16.00</td>
<td>1.79</td>
<td>16.57</td>
<td>10.80</td>
<td>10.80</td>
<td>9.44</td>
<td>9.77</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>16.00</td>
<td>0.57</td>
<td>16.57</td>
<td>16.83</td>
<td>25.49</td>
<td>1.22</td>
<td>14.89</td>
</tr>
<tr>
<td>4</td>
<td>Case 1</td>
<td>17.78</td>
<td>2.26</td>
<td>16.57</td>
<td>10.80</td>
<td>10.80</td>
<td>12.14</td>
<td>10.40</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>17.78</td>
<td>0.75</td>
<td>16.57</td>
<td>16.83</td>
<td>25.49</td>
<td>1.61</td>
<td>16.32</td>
</tr>
</tbody>
</table>

Table 3.4: OpenCL GPU @ 800.0 MHz run-time in milliseconds.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Test Case</th>
<th>AbsDiff</th>
<th>Mean Stddev</th>
<th>Binary</th>
<th>Erode</th>
<th>Dilate</th>
<th>Count</th>
<th>Read</th>
<th>Write</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8</td>
<td>Case 1</td>
<td>160</td>
<td>115</td>
<td>116</td>
<td>443</td>
<td>443</td>
<td>286</td>
<td>32</td>
<td>1595</td>
<td>12374</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>160</td>
<td>171</td>
<td>116</td>
<td>1831</td>
<td>9706</td>
<td>358</td>
<td>32</td>
<td>1595</td>
<td>12374</td>
</tr>
<tr>
<td>float32</td>
<td>Case 1</td>
<td>160</td>
<td>205</td>
<td>132</td>
<td>784</td>
<td>784</td>
<td>330</td>
<td>33</td>
<td>2428</td>
<td>25214</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>160</td>
<td>221</td>
<td>132</td>
<td>3684</td>
<td>20631</td>
<td>353</td>
<td>33</td>
<td>1517</td>
<td>7495</td>
</tr>
<tr>
<td>float32[4]</td>
<td>Case 1</td>
<td>162</td>
<td>43</td>
<td>132</td>
<td>531</td>
<td>531</td>
<td>85</td>
<td>33</td>
<td>2428</td>
<td>25214</td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>162</td>
<td>58</td>
<td>132</td>
<td>1565</td>
<td>5455</td>
<td>90</td>
<td>33</td>
<td>1517</td>
<td>7495</td>
</tr>
</tbody>
</table>
Table 3.5: OpenCV CPU @ 1.0 GHz run-time in milliseconds.

<table>
<thead>
<tr>
<th>CPU Cores</th>
<th>Test Case</th>
<th>AbsDiff</th>
<th>Mean</th>
<th>Stddev</th>
<th>Binary</th>
<th>Erode</th>
<th>Dilate</th>
<th>Count</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Case 1</td>
<td>16</td>
<td>101</td>
<td>13</td>
<td>57</td>
<td>57</td>
<td>30</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>16</td>
<td>535</td>
<td>13</td>
<td>135</td>
<td>315</td>
<td>309</td>
<td>1323</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Case 1</td>
<td>11</td>
<td>52</td>
<td>11</td>
<td>57</td>
<td>57</td>
<td>16</td>
<td>204</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>11</td>
<td>227</td>
<td>11</td>
<td>135</td>
<td>315</td>
<td>155</td>
<td>854</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Case 1</td>
<td>11</td>
<td>35</td>
<td>9</td>
<td>57</td>
<td>57</td>
<td>11</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>11</td>
<td>148</td>
<td>9</td>
<td>135</td>
<td>315</td>
<td>105</td>
<td>723</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Case 1</td>
<td>10</td>
<td>27</td>
<td>8</td>
<td>57</td>
<td>57</td>
<td>9</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>10</td>
<td>110</td>
<td>8</td>
<td>135</td>
<td>315</td>
<td>78</td>
<td>656</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: OpenCV CPU @ 1.5 GHz run-time in milliseconds.

<table>
<thead>
<tr>
<th>CPU Cores</th>
<th>Test Case</th>
<th>AbsDiff</th>
<th>Mean</th>
<th>Stddev</th>
<th>Binary</th>
<th>Erode</th>
<th>Dilate</th>
<th>Count</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Case 1</td>
<td>14</td>
<td>69</td>
<td>9</td>
<td>41</td>
<td>41</td>
<td>21</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>14</td>
<td>304</td>
<td>9</td>
<td>93</td>
<td>214</td>
<td>219</td>
<td>853</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Case 1</td>
<td>11</td>
<td>35</td>
<td>8</td>
<td>41</td>
<td>41</td>
<td>12</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>11</td>
<td>153</td>
<td>8</td>
<td>93</td>
<td>214</td>
<td>111</td>
<td>590</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Case 1</td>
<td>10</td>
<td>24</td>
<td>7</td>
<td>41</td>
<td>41</td>
<td>9</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>10</td>
<td>102</td>
<td>7</td>
<td>93</td>
<td>214</td>
<td>74</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Case 1</td>
<td>9</td>
<td>19</td>
<td>7</td>
<td>41</td>
<td>41</td>
<td>7</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Case 2</td>
<td>9</td>
<td>77</td>
<td>7</td>
<td>93</td>
<td>214</td>
<td>56</td>
<td>456</td>
<td></td>
</tr>
</tbody>
</table>
Other i.MX 8 processors exist that offer more capabilities such as more CPU cores and a more powerful GPU. If more processing capabilities are required, another one of the i.MX 8 processors could be used, and the results could be interpolated to estimate performance. The i.MX 8M Quad SOC has similar CPU specs to the 64-bit Raspberry Pi boards. Therefore, they were not considered for this evaluation. Many QualComm, Samsung, and other cellular phone-oriented processors would likely be ideal for this task. However, the i.MX 8 family of processors has similar specs, and the cellular phone processors have many features that were not required by the application. Other interesting options do exist for solving the low-power processing problem. The Intel Movidius NCS-2 boasts inference of convolutional neural networks with < 1 W of power consumption. While the chip is optimized for neural network inference, it could possibly be used to accelerate image processing tasks. The Nvidia Jetson TX-2 was also a viable option. It would likely be able to process images at a much higher frame rate, but the power consumption would be too much for the application.

3.4 Conclusions

Low-power, long-range, wide-FOV image detection, tracking, and classification is a difficult problem as it requires a lot of processing capabilities and requires low-power hardware to be able to run on battery power for extended periods of time. Many state-of-the-art CPUs and hardware accelerators, such as GPUs and FPGAs, exist that lend themselves well to this problem.
A proposed solution to this problem was evaluated using the NXP i.MX 8 family of SOCs. The SOCs include ARM 64-bit processors paired with Vivante GPUs. An image processing chain using the selected hardware was created that was made up of simple tasks that could be performed quickly by limited hardware accelerators. Using unsigned 8-bit integer math, the ARMv8 Neon co-processor proved to be a very powerful image processing hardware accelerator: more so than the Vivante GPU. However, if more complicated image processing was required that used floating-point math, the Vivante GPU would likely be very beneficial.

The proposed image processing chain is able to create AOIs but cannot accurately determine the exact location of the object of interest, nor the type of object present. The proposed solution is also susceptible to close-range moving clutter as large number of pixels can change causing false alarms. Therefore, the processing chain can be considered a pre-processing step. Further object classification could be done with a convolutional neural network. Real-time processing of large images using a convolutional neural network would require very large amounts of processing, but feeding smaller, pre-processed, cropped images to a simple convolutional neural network is feasible in a low-power environment.
4.1 Conclusions

Low-power, long-range, wide-FOV image detection, tracking, and classification is a difficult problem as it not only requires significant processing capabilities but also must be able to operate on a constrained power budget. Many state-of-the-art CPUs and hardware accelerators, such as GPUs, FPGAs, and ASICs, exist that lend themselves well to this problem.

A proposed solution to this problem was evaluated using the NXP i.MX 8 family of SOCs. The SOCs include ARM 64-bit processors paired with Vivante GPUs. An image processing chain using the selected hardware was created that was made up of simple tasks that could be performed quickly by limited hardware accelerators. Using unsigned 8-bit integer math, the ARMv8 Neon co-processor proved to be a very powerful image processing hardware accelerator, more so than the Vivante GPU. However, if more complicated image processing was required that used floating point math, the Vivante GPU would likely be very beneficial.

The proposed image processing chain is able to create AOIs but cannot accurately determine the exact location of the target, nor the type of target present. The proposed solution is also susceptible to close-range moving clutter as large number of pixels can
change introducing false alarms. Therefore, the processing chain can be considered a pre-
processing step. Further target classification could be done with a convolutional neural
network. Real-time processing large images using a convolutional neural network would
require very large amounts of processing, but feeding smaller, pre-processed, cropped im-
ages to a simple convolutional neural network is feasible in a low-power environment.

4.2 Contributions

The goal of this work was to define a method for detecting targets in images at longer
ranges while maintaining a wide-FOV on hardware that is capable of running on battery
power for extended periods of time. An image differencing method was defined for per-
foming the task that could run efficiently on a low-power system. The algorithm require-
ments for different applications will change though, so I focused more on the hardware
implementation aspects. It is shown that processors designed for mobile phones or sim-
ilar applications with 1.5 W power consumption are capable of performing these tasks,
especially with many of the recent advances in Silicon manufacturing and architecture.

4.3 Future Work

I plan to pursue further work on low-power image classification using convolutional
neural networks. This area has been of interest to me, because it is very useful for small,
battery powered devices. Recent advances in the ARMv8.2 instruction set has added a
native SIMD dot product instruction that could greatly improve neural network inference
time as the primitive convolution operation in a convolutional neural network can be de-
composed to dot product operations. Tight integration with 16-bit floating capable newer,
low-power Mali GPUs could also improve the capabilities of ARM SOCs to perform neural network inference while using only a few watts of power. ARMv8.2 capable processors have recently been made available for mobile phones but have not hit other commercial markets. Other companies such as Intel have also produced low-power hardware for neural network inference such as the Intel Myriad vision processors which claim less 1 W power consumption for real-time convolutional neural network inference. As these new hardware accelerators become available, I hope to understand their capabilities and limitations so that I can further field of low-power image detection, tracking, and classification.
BIBLIOGRAPHY


