Title
Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs

Permalink
https://escholarship.org/uc/item/82r211bk

Journal
IEEE Transactions on Circuits and Systems for Video Technology, 25(6)

ISSN
1051-8215

Authors
Ma, X
Najjar, WA
Roy-Chowdhury, AK

Publication Date
2015-06-01

DOI
10.1109/TCSVT.2014.2360030

Peer reviewed
Evaluation and Acceleration of High-Throughput Fixed-Point Object Detection on FPGAs

Xiaoyin Ma, Student Member, IEEE, Walid A. Najjar, Fellow, IEEE, Amit K. Roy-Chowdhury, Sr. Member, IEEE

Abstract—The reliance on object or people detection is rapidly growing beyond surveillance to industrial and social applications. The Histogram of Oriented Gradients (HOG), one of the most popular object detection algorithms, achieves high detection accuracy but delivers just under one frame-per-second (fps) on a high-end CPU. FPGA accelerations of this algorithm are limited by the intensive floating-point computations. All current fixed-point HOG implementations use large bit-width to maintain detection accuracy, or perform poorly at reduced data precision. In this paper we introduce the full-image evaluation methodology to explore the FPGA implementation of HOG using reduced bit-width. This approach lessens the required area resources on the FPGA and increases the clock frequency and hence the throughput per device through increased parallelism. We evaluate the detection accuracy of the fixed-point HOG by applying state-of-the-art computer vision pedestrian detection evaluation metrics and show it performs as well as the original floating-point code from OpenCV. We then show our single FPGA implementation achieves a 68.7x higher throughput than a high-end CPU, 5.1x higher than a high-end GPU, and 7.8x higher than the same implementation using floating-point on the same FPGA. A power consumption comparison for different platforms shows our fixed-point FPGA implementation uses 130x less power than CPU, and 31x less energy than GPU to process one image.

Index Terms—Computer vision; fixed-point; pedestrian detection; histogram of oriented gradients;

I. INTRODUCTION

Like many applications relying on numeric computations, computer vision applications make extensive use of floating-point number representation, both single and double precision. The major advantage of floating-point representation is the very large range of values that can be represented with a limited number of bits. Most CPU, and all GPU designs have been extensively optimized for short-latency high-throughput processing of floating-point operations. On an FPGA, the bit-width of operands in an application is a major determinant of its resource utilization, the achievable clock frequency and hence its throughput. By using a fixed-point representation with fewer bits, an application developer could implement more processing units on a given FPGA and each unit could achieve a higher-clock frequency because of its smaller footprint. However, smaller bit-width may lead to inaccurate or incorrect results.

Object and human detection are fundamental problems in computer vision and a very active research area. In these applications a high throughput and an economy of resources are highly desirable features allowing the applications to be embedded in mobile or field-deployable equipment. The HOG algorithm [1], developed for human detection and expanded to other object detection, is one of the most successful and popular algorithms in its class. In this algorithm, object descriptors are extracted from detection window with grids of overlapping blocks. Each block is divided into cells in which histograms of intensity gradients are collected as HOG features. Vectors of histograms are normalized and passed to a Support Vector Machine (SVM) classifier [2], [3] to recognize a person or an object.

In this paper we explore the effects of reduced bit-width on the accuracy and performance of the HOG object detection algorithm implemented on an FPGA by applying the full-image evaluation methodology and state-of-the-art computer vision pedestrian detection metrics. Using four sets of benchmarks, totaling 10,000 frames, we show that reducing the bit-width to 13-bits preserves the same detection accuracy as the original floating-point. We describe an FPGA implementation of the HOG algorithm and explore the impact of reduced data precision on the area and clock frequency of the design. The throughput of the 13-bit fixed-point design on a single FPGA is then compared to that on CPU using floating-point (68.7x), a CPU with the Intel IPP library (60x), a high-end GPU (5.1x) and the same FPGA design using floating-point data (7.8x). This paper is built on our previous work [4] that evaluates the HOG detection accuracy under reduced bit-width. In this paper, we study the detection accuracy based on the precision and recall values across four different benchmarks to demonstrate the choice of 13-bits fixed-point over other bit-width. Moreover, we compare our HOG-Engine FPGA resource utilization between fixed-point and floating-point, and different bit-width fixed-point implementations. Additionally, a two-stage processing architecture is proposed to boost the throughput of our original single HOG-Engine from 411 ms per image to 44 ms per image (9.3x better performance). Furthermore, the system architecture and memory interface are discussed in this paper. Last but not least, we include the power consumption comparison for different platforms.

The contributions of this paper are (1) A systematic experimental evaluation of the detection accuracy of the HOG algorithm with fixed-point data using full-image evaluation as opposed to traditional per-window evaluation while varying the bit-width, using 10,000 benchmark frames with known ground truth. (2) A fully pipelined, two-step FPGA implementation of HOG on a Xilinx Virtex-6 LX760 FPGA attached to Convey HC-2ex computer; its architecture occupies 6.5% of...
the FPGA resources. (3) A comparison of the throughput on FPGA, fixed and floating-point, CPU, with and without Intel IPP library, and the Nvidia Tesla K20 GPU, using 640×480 images at 1.05 scale factor, with bilinear interpolation and a window stride of four pixels (a low scale factor or window stride increases the detection accuracy but also the computational load). (4) A power consumption comparison between CPU, GPU and FPGA.

The remainder of this paper is organized as follows: Section II covers related work in the hardware acceleration of HOG detection algorithms as well as the tradeoffs between fixed and floating-point representations on FPGAs. Section III offers a detailed description of the HOG algorithm. The fixed-point implementation of HOG is evaluated in Section IV using four benchmarks and we demonstrate that the 13-bits representation is optimal across all benchmarks. Our FPGA architecture of HOG is evaluated in Section IV using four pedestrian detection benchmarks totaling 10,000 frames with known ground truth. Finally we constructed our fully pipelined FPGA accelerator and compared the throughput to those on state-of-the-art GPU and CPU.

Many hardware accelerated solutions have been proposed for HOG pedestrian detection, mostly using GPUs, with a reported speed-up of up to 67x [10], [11], [12], [13], [14], [15]. Because of deeply pipelined architectures and lower power consumption, FPGA platforms often provide higher execution speed and better energy efficiency over GPUs [16]. An FPGA-GPU hybrid system was proposed in [17] using FPGA to extract HOG features and GPU to perform classification; it achieved a throughput of 10,000 detection windows per second for FPGA execution. Note that whole images (frames) were not tested.

In [18] a HOG feature extractor circuit for pedestrian and vehicle detection, using fixed-point data, was described with an estimated throughput of 33 fps at a single scale for 640×480 images. The detection accuracy was not reported or compared to a reference implementation. In [19], a HOG detection system was implemented on an Altera Stratix II FPGA using window size of 16×32 and scale factor of 1.2 achieving an estimated 30 fps for 640×480 video. Our experiments have shown that a scale factor 1.2 has 3.25x less computation than the 1.05 scale factor used in this paper and 6x poorer detection accuracy, in terms of true positives. In [16] a person detection execution on CPU, GPU and FPGA was compared for power, speed and accuracy. The FPGA implementation focused only on 4 out of 37 scales for 640×480 images and achieves 30 fps. In none of the papers above was the reduced bit-width used for HOG detection. A pedestrian detection system processing 18 scales of 1920×1080 resolution images at 64 fps was reported in [20]. Its throughput was estimated via simulation.

In [21], a real-time person detection was implemented on FPGA with a 62.5 fps on images with size equivalent to 320×240 at a single scale. While the data range of fixed-point values was reported (8-bits for input pixel, 19-bits for gradient, 14-bits for each histogram, and 33-bits for normalized histogram), there was no exploration of the tradeoffs in detection accuracy with reduced bit-width. Moreover, their fixed-point implementation showed a decrease in detection accuracy.

Mizuno et al. [22] have reported on the fixed-point parameter optimization by comparing the per-window detection results with the ground truth for INRIA person dataset [11]. The difficulty of INRIA benchmark is much simpler than those used in this paper. A fixed-point version of HOG detection for a digital signal processor (DSP) PICTOR was discussed in [23]. The detection accuracy was compared to with MATLAB’s double-precision code without, however, reporting the bit-width. These approaches focused on comparing the detection window level output difference between fixed-point and floating-point computations. Nevertheless, per-window based evaluation methodology can fail to represent full image performance. For example many detected false positive windows in window-based evaluation can be removed by merging nearby bounding boxes in post-processing as shown in [8].

Scale factor (the ratio to scale the image after each detection) and window stride are one of the most important param-
TABLE I: Comparison of parameters and performance for various FPGA and GPU implementations.

<table>
<thead>
<tr>
<th>Scale</th>
<th># scales</th>
<th># bins</th>
<th>Win. stride</th>
<th>Win./frame</th>
<th>Resolution</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our</td>
<td>1.05</td>
<td>38</td>
<td>9</td>
<td>4</td>
<td>541,10 540 / 480</td>
<td>68.18</td>
</tr>
<tr>
<td>64 DSPs</td>
<td>1.1</td>
<td>13</td>
<td>9</td>
<td>8</td>
<td>27,500 1280 / 1024</td>
<td>64</td>
</tr>
<tr>
<td>1.2</td>
<td>13</td>
<td>9</td>
<td>20,568 1024 / 768</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>1</td>
<td>9</td>
<td>5,780 960 / 600</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>1</td>
<td>9</td>
<td>1,280 480 / 480</td>
<td>62.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>1</td>
<td>1</td>
<td>1,000 800 / 800</td>
<td>&gt;10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>&gt;1</td>
<td>8</td>
<td>56,666 640 / 480</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>&gt;1</td>
<td>8</td>
<td>3,915 320 / 240</td>
<td>38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE II: FPGA resource utilization of arithmetic operations between fixed-point and floating-point data.

<table>
<thead>
<tr>
<th># Bits</th>
<th>Reg. LUTs</th>
<th>DSPs</th>
<th>Latency</th>
<th>F (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point Addition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>130</td>
<td>16</td>
<td>2</td>
<td>235</td>
</tr>
<tr>
<td>32</td>
<td>66</td>
<td>8</td>
<td>2</td>
<td>541</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
<td>20</td>
<td>2</td>
<td>627</td>
</tr>
<tr>
<td>13</td>
<td>18</td>
<td>15</td>
<td>2</td>
<td>609</td>
</tr>
<tr>
<td>Floating-Point Addition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1034</td>
<td>800</td>
<td>12</td>
<td>268</td>
</tr>
<tr>
<td>32</td>
<td>541</td>
<td>397</td>
<td>12</td>
<td>390</td>
</tr>
<tr>
<td>16</td>
<td>224</td>
<td>171</td>
<td>8</td>
<td>397</td>
</tr>
<tr>
<td>13</td>
<td>193</td>
<td>142</td>
<td>8</td>
<td>412</td>
</tr>
<tr>
<td>Fixed-Point Multiplication without DSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>4296</td>
<td>2593</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>1098</td>
<td>1099</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>279</td>
<td>283</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>216</td>
<td>194</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Floating-Point Multiplication without DSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2431</td>
<td>2390</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>32</td>
<td>661</td>
<td>654</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>202</td>
<td>185</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>151</td>
<td>129</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Fixed-Point Multiplication with DSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>859</td>
<td>437</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>53</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>473</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Floating-Point Multiplication with DSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>391</td>
<td>308</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>32</td>
<td>179</td>
<td>132</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>89</td>
<td>74</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>80</td>
<td>64</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

IEEE standard. 13-bit floating-point has five exponent bits and eight fraction bits. As shown in Table I, fixed-point additions use 10.5-12.5x less LUTs than floating-point addition while operating at 1.6-2.4x higher frequency. Furthermore, floating-point additions require more registers as the computation takes several clock cycles.

Fixed-point multiplication requires more FPGA area than floating-point multiplication since the product of two 32-bit integer multiplication is 64-bit while the result of two 32-bit floating-point multiplication will yield another 32-bit value, as shown in Table II. However, the multiplication of small bit-width values can take the advantage of on-chip DSP blocks to ease the area usage. Table II shows the FPGA resource utilization when using DSP block for both fixed-point and floating-point multiplications. 32-bit and below fixed-point multiplication benefit from the usage of DSP blocks.

Fixed-point arithmetic uses less FPGA area and runs at a higher frequency than floating-point operations. Hence, with sufficient memory bandwidth, one can place more fixed-point modules on a single FPGA running at higher frequency to increase the overall throughput. However, the use of fixed-point data may compromise the accuracy of the detection. The original HOG algorithm uses single-precision floating-point for all computations. Replacing the large range floating-point data with fixed-point value may potentially cause data overflow. To further increase the computation throughput, we use 13-bit fixed-point data as the input for the fixed-point implementation in order to achieve the same level of accuracy as the floating-point implementation. The latency for a regular fixed-point adder should be one. An additional output stage is intentionally added here to obtain correct timing results.

1Simulation estimated speed, no actual implementation.
want to use the least possible number of bits for each step but the use of lower bit-width values may introduce uncertainties in the final classification result. To find the exact bit-width that can be used in fixed-point detection, we carefully evaluate every computation step of the HOG pedestrian detection implementation in OpenCV [9]. We determine that 27-bit fixed-point is sufficient to maintain a similar precision as the original floating-point data representation. The exact bit-width for each data is discussed in Section IV. Then we construct our own HOG detection program that performs the exact detection procedure using fixed-point data. We then gradually decrease the number of bits, starting from 27 bits, and compared the detection outcome with the original floating-point OpenCV detection to find the least possible number of bits suitable for HOG detection. The detailed comparison process is described in Section IV.

### III. Histograms of Oriented Gradients

In the HOG algorithm object descriptors are extracted from detection window with grids of overlapping blocks. Each block is divided into cells in which histograms of intensity gradients are collected as HOG features. Vectors of histograms are normalized and passed through detector for detection. The detailed detection algorithm is described below.

#### A. Orientation and Magnitude Computing

Input pixel values are converted to gradients in HOG-based object detection. As shown in Equation 1, the gradient of
dx = \text{pixel}(x + 1, y) - \text{pixel}(x - 1, y)
dy = \text{pixel}(x, y + 1) - \text{pixel}(x, y - 1)

(1)
pixels, \(dx\) and \(dy\) are obtained by using a simple 1-D mask
\[
\begin{bmatrix}
1 & 0 & -1
\end{bmatrix}
\]

Then, for each pair of \(dx\) and \(dy\), the magnitude \(m(x, y)\) and orientation \(\theta(x, y)\) are computed as Equation 2

\[
\begin{align*}
m(x, y) & = \sqrt{dx^2 + dy^2} \\
\theta(x, y) & = \text{atan}\left(\frac{dy}{dx}\right)
\end{align*}
\]

(2)

For colored images, the magnitudes are computed for each individual channel and the one with largest magnitude value is chosen.

#### B. Histogram Generation

In this algorithm, every 8 \times 8 pixels form a cell, and every 2 \times 2 cells form a block, as illustrated in Figure 1. The magnitudes are binned into histograms based on the orientations within each cell. Figure 2 shows the binning diagram used in HOG. Each cell generates a 9-bin histogram for orientation in the range of 0\(^\circ\) – 360\(^\circ\). The orientations are "unsigned" meaning that from 180\(^\circ\) – 360\(^\circ\) the binning are the same as 0\(^\circ\) – 180\(^\circ\). The bin value is updated by weighted magnitude value. The magnitude weight is based on the difference between the angle and bin edge as shown in Equation 3 (floor function is used to compute bin edge).

\[
\alpha = \frac{9 \cdot \theta}{\pi} - \text{floor}\left(\frac{9 \cdot \theta}{\pi} - 0.5\right)
\]

(3)

In addition, the bin after current bin will also be updated to reduce aliasing as shown in Equations 3 and 4 (\(vote_0\) is for current bin, \(vote_1\) is for the next bin). Furthermore, each vote in a cell is bilinearly interpolated to the neighboring cell. Finally, a Gaussian filter is applied to each vote based on its location within a block to mitigate the contribution of pixels close to the block edge. Thus, the final votes can be written as the products of the vote and two weights (\(weight_{\text{intrpl}}\), \(weight_{\text{gauss}}\)) as in Equation 5. Histograms within a block are concatenated together forming a 1 \times 36 vector. All vectors in a sliding window are also concatenated as the final descriptor vector. Therefore, a 48 \times 96-pixel window (Figure 1) has 5 \times 11 blocks with a total of 1980 histograms (a 1 \times 1980 vector).

#### C. Histogram Normalization and SVM Classification

Block histograms are normalized to minimize the effect of local illumination variance and foreground-background contrast. The block histogram vector is normalized twice using Equation 6. In general, each vector element is divided by the vector’s Euclidean length (square root of elements’ sum of squares). Constant value \(c\) is used to avoid division by zero. In the first normalization, \(c\) value is 3.6 and the maximum value for each element is limited to 0.2 after normalization. Then,

\[
V = \frac{\vec{v}}{\sqrt{|\vec{v}|^2 + c}}
\]

(6)
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSVT.2014.2360030, IEEE Transactions on Circuits and Systems for Video Technology

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. , NO. , XXX,XXXX 5

A. Implementation of Fixed-Point HOG Detection

For the implementation of fixed-point HOG pedestrian detection, we started with the Daimler detector (a pre-trained SVM classifier) came with OpenCV [9], [27] with a window size of 48 × 96 pixels. The window stride is 4 pixels for both horizontal and vertical direction. Moreover, we choose the final threshold as 0.5 (only when $s > 0.5$, the window is considered as positive) to limit the total number of positive windows. All other parameters discussed in Section III, e.g., trained classifier vector values, are converted to fixed-point data for detection.

The implementation includes all the steps of HOG detection: from the initial orientation and magnitude computation to the computing of the final score $s$. The final grouping algorithm (combine multiple detection windows at various scales into a single rectangle) is not included. For an $n$-bit fixed-point implementation, the bit-width of individual parameters are shown in Figure 3. As we reduce the bit-width, all intermediate values are scaled accordingly, as shown in Figure 3. However, the sum of histogram squares in Equation 6 (denominator part without computing square root) for the first normalization has a very large data range. Thus it will remain 27 bits with 0 fractional bits for $n$ is 16 or lower. All constant parameters in HOG detection are converted to fixed-point using 0 integer bits and $n$ fractional bits, as discussed in Section III. Also the interpolation and Gaussian weights (Equation 5) are combined into a single value before converting to fixed-point.

B. Evaluation Methodology

Traditionally, fixed-point arithmetic implementation focuses on the absolute errors introduced by the reduced bit-width. Specifically in object detection, both fixed-point and floating-point object detectors are applied to detection windows known as object or background for detection rate comparison. The desired bit-width is determined by the minimum acceptable detection rate using certain fixed-point bit-width. However, this approach may not correctly predict the actual detection performance when considering the entire frame across multiple image scales. Usually a post-processing step is performed on all positive windows across the image at all scales to merge nearby positive windows. This step can reduce the number of false positive windows found by the detector. On the other hand, it can introduce detection errors such as incorrectly detected object sizes that would otherwise not have been found in window-based evaluation. Thus, to evaluate the effect of reduced data precision, methods other than window-based evaluation are needed. Dollar et al. [28], [8] proposed the per-image evaluation approach as opposed to per-window methodology for pedestrian detection algorithm evaluation. They reported the classification performance of various classifiers for these two approaches. In general, the per-image based approach is more meaningful as well as practical. Therefore, we applied this method in our fixed-point detection to find the optimal bit-width. The detailed evaluation results will be discussed in Section IV-D.
(e) Overall Detection Results

Caltech Benchmark Results

Precision and Recall values

Precision
Recall

0.44
0.54

0.54
0.64

0.46
0.56

0.56
0.66

0.40
0.50

0.60

0.60

0.58

0.58

0.43

0.35

0.45

0.55

0.47

0.45

0.62

0.62

8 

BB

the commonly accepted PASCAL method shown in Equation

III. To match our detection results to the ground truth we use

pedestrian objects we use for evaluation are shown in Table

the detection window height). The number of images and

images are

JELMOLI, and SUNNY DAY sequences) [29]. All benchmark

[27], TUD-Brussels [26], Caltech Pedestrian Detection [28],

point HOG detection: Daimler Mono Pedestrian Detection

C. Benchmarks and Detection Evaluation

We have used four benchmarks to evaluate our fixed-point HOG detection: Daimler Mono Pedestrian Detection [27], TUD-Brussels [26], Caltech Pedestrian Detection [28], and three sequences from ETH datasets (the BAHNHOF, JELMOLI, and SUNNY DAY sequences) [29]. All benchmark images are 640 × 480 and have a ground truth of pedestrians. Every ground truth pedestrian is marked by a rectangular bounding box (BB), indicating its location and size. In our evaluation, we selected the frames that contain at least one pedestrian object with a BB height > 67 pixels (70% of the detection window height). The number of images and pedestrian objects we use for evaluation are shown in Table III. To match our detection results to the ground truth we use the commonly accepted PASCAL method shown in Equation [8] [30]. BB_{det} refers to the BB from the detection and BB_{grt} is the ground truth BB. Two objects are matched when their overlapping area is more than 50% of the union area. Each detected object may be matched at most once to a ground truth object. In addition, adjustments for both BB_{det} and BB_{grt} are made based on the methods described in [23], [8]. For each BB_{grt}, the aspect ratio of a rectangle depends on the limb position of a walking pedestrian. Thus, all BB_{grt} are resized to an aspect ratio of 0.41 by keeping the center of the object. What’s more, each BB_{det} corresponds to a detection window of 48 × 96 pixels with about twelve-pixel paddings on top and bottom of each pedestrian [27]. Therefore, the BB_{det} height is resized by a scale of 0.78125, then the aspect ratio is resized to 0.41. These processes provide better matching between ground truth and detection result. Moreover, ground truth objects near the image edge, with height below 67 pixels and non-pedestrian are set to ignore. Ignored objects are not counted as true positive if matched and will not contribute to false negatives if unmatched.

D. Evaluation Result

For fixed-point detection, we perform detection in 27-bits down to 11-bits. The number of bits for each fixed-point detection is shown in Figure 3 (substitute n with corresponding bits). In addition to the single-precision floating-point and fixed-point detection, we construct another detection with fixed-point detection, we construct another detection with

TABLE III: Number of frames and objects for each benchmark sequence after filtering.

<table>
<thead>
<tr>
<th></th>
<th>Daimler</th>
<th>Caltech</th>
<th>TUD-Brussels</th>
<th>ETH</th>
</tr>
</thead>
<tbody>
<tr>
<td># image</td>
<td>2117</td>
<td>5346</td>
<td>237</td>
<td>1785</td>
</tr>
<tr>
<td># object</td>
<td>2603</td>
<td>8310</td>
<td>661</td>
<td>8076</td>
</tr>
</tbody>
</table>

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. , NO. , XXX,XXXX 6
We show the OpenCV detection (cv-float), double precision floating-point (double) and a subset of fixed-point detection results for each benchmark and the overall results by averaging the four individual benchmark, in Figure 4. Detection precision and recall are computed using Equation 9.

\[
\text{precision} = \frac{TP}{TP + FP} \\
\text{recall} = \frac{TP}{TP + FN}
\]  

(9)

Fixed-point detection results from 27-bits to 18-bits are almost identical to the floating-point results in all benchmarks. For 17-bits and lower, detection at lower bits generally increases recall and decreases precision. For all benchmarks, we observe an increase of precision in fix-17, and a decrease in recall. This is due to a slight decline of the TP, but a significant reduction in FP. Besides, the reduced TP also results the contraction of FN, hence the loss of recall as shown in Figure 4 (a), (b), (d) for fix-17. The overall recall is increased from 0.458 at cv-float to 0.465 for fix-13 and 0.475 for fix-12 while the precision grows from cv-float’s 0.561 to 0.564 for fix-13 and dropped to 0.555 at fix-12. Moreover fix-11 has boosted recall to 0.488 with a significant decrease of precision to 0.522.

Finally, we choose 13-bits in our hardware implementation as it provides a balance between precision and recall and consistent performance across all benchmarks in per-image evaluation. The detailed hardware implementation is discussed in Section V.

V. FPGA IMPLEMENTATION

In this section we describe and evaluate the fixed and floating-point implementations of the HOG detection on a FPGA. We compare the throughput to those of the CPU and GPU implementations.

A. FPGA Platform

We implement our HOG detection system on the Convey HC-2ex machine [31]. The system’s hybrid-core architecture is composed of two Intel Xeon E5-2643 four-core processors and four Xilinx Virtex-6 LX760 FPGAs. Both CPUs and FPGAs share a globally addressable 256 GB virtual memory, 128 GB on FPGA side and 128 GB on CPU side. FPGA memory is connected to CPUs via one PCIe 3.0 x16. The FPGA memory system is built around Convey’s Scatter-Gather DIMMs to provide random transfer of 8-byte bursts at near peak bandwidth [32]. All FPGAs are linked to host processors through an Application Engine Hub that can send and transfer opcodes and scalar operands to FPGA. Each FPGA has 16 64-bit memory channels at 150 MHz controlled by eight memory controllers. The FPGA program runs at 150 MHz.

The memory subsystem provides a highly parallel and high bandwidth (19.2 GB/s per FPGA) connection between FPGAs and physical memory. These properties permit the user to design complicated memory access pattern in the HOG-Engine to achieve maximum performance. The hardware architecture and memory accesses will be discussed in the following sections.

The host software is written in C++ and the FPGA code is developed in Verilog. The design is simulated using Convey Personality Development Kit and Modelsim Foreign Language Interface for hardware and software co-simulation. Synthesis is performed using Xilinx ISE 14.3. We have used Xilinx Core Generator to generate fixed-point multiplication and square root IP cores. For fixed-point division, we use the divider from [33]. Each HOG-Engine uses 138 fixed-point multiplication modules. To ease the FPGA timing, we implement 64 multiplication modules in bilinear interpolation of votes and four multiplication for magnitude voting on DSPs (a total of 68 DSP slices per HOG-Engine), others on LUTs. The normalization module is implemented by using square root, division and multiplication modules. First the histogram squares are summed, then sent to square root module. Finally the reciprocal is computed by the divider core. The normalized histogram value is the multiplication of histogram and the reciprocal value.

B. HOG-Engine Architecture

As a first step, we profiled the HOG pedestrian detection code, on CPU, to find the most critical computation in HOG detection. The profiling information is shown in Table IV. Post processing is used in all object detection algorithms to combine similar windows into one. For our HOG-Engine, we focused on implementing the most computational expensive parts of HOG detection on FPGA: orientation binning, magnitude voting, histogram generation, normalization, and SVM classification. All other computations are performed in software.

Our implementation design on FPGA consists of two steps: histogram generation and classification. Histogram generation produces weighted votes, accumulates them in cell histograms that are combined to form block histograms. Block histograms are then normalized twice and sent back to memory. The classification module fetches the normalized histograms from memory and performs SVM classification to generate the final score. The schematics of our HOG-Engine for histogram generation and classification are shown in Figure 5.

The HOG-Engine reads the magnitude and orientation values from memory. Each pair of magnitude and orientation values is packed into a single 32-bit integer. As one memory access returns a 64-bit value, two pairs of magnitude and orientation values are returned in a single memory access. The HOG-Engine fetches pixels from two rows of cells alternately to increase parallelism as shown in Figure 5a. For each pair of orientation and magnitude, two vote values (vote0, vote1) and a bin number are computed. As discussed in Section II-B each cell has 64 pixels and generates a 9-bin histogram. However, due to bilinear interpolation, each vote is weighted and interpolated into all other cells in the same block. To reduce the interaction between different cells, each cell produces 4 \times 9-bin histograms. In addition, as a cell could be in one of the four positions in a block shown in Figure 5a (TL, TR, BL, BR), in our implementation, a single cell will generate 4 \times 36-histograms. Cell histograms are then combined using a simple vector add to obtain the block histogram when all four cells in a block are processed. Unused cell histograms are automatically discarded based on
TABLE IV: HOG detection profiling result.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization &amp; read image</td>
<td>0.65</td>
</tr>
<tr>
<td>Image resize</td>
<td>0.65</td>
</tr>
<tr>
<td>Magnitude &amp; angle</td>
<td>0.62</td>
</tr>
<tr>
<td>Binning &amp; voting</td>
<td>3.44</td>
</tr>
<tr>
<td>Block hist. gen. &amp; norm.</td>
<td>46.20</td>
</tr>
<tr>
<td>SVM</td>
<td>18.32</td>
</tr>
<tr>
<td>Post processing</td>
<td>29.63</td>
</tr>
</tbody>
</table>

The aforementioned architecture works well with a window stride of eight pixels (one cell). However, to further improve the detection accuracy, we would like to use a stride of four pixels (half cell). When the window stride is four pixels, all cells and blocks in the new window are changed and we cannot re-use previously computed cell histogram results. To solve this problem, we treat a single scale of image as four sub-scales, processing each with a window stride of eight pixels. Concretely, we first process the sub-scale starting at the first column, first row of pixels using the above HOG-Engine. Then, we process the same image again starting at the first row, fifth column. Thirdly, we compute histograms starting at the fifth row, first column, and finally fifth row, fifth column. Therefore, a total of 34 scales image is divided into 134 sub-scales (the last scale only have two sub-scales). This design allows us to use the same HOG-Engine architecture to efficiently generate histograms and perform classification.

C. Input/Output Controller

Both histogram generation and classification modules have input and output controllers to interface with the FPGA memory system. As discussed previously, the HOG-Engine processes a frame at 34 different scales. In addition, each scale is divided into four sub-scales to slide detection window by four pixels vertically and horizontally. These controllers are responsible to access images at different scales. The image is resized in software and then magnitude and orientation are computed. For a single frame, magnitude and orientation values at 34 scales are concatenated into a single array and sent to FPGA memory. Histogram generation input controller generates pixel (magnitude and orientation pairs) addresses by using three nested state machines, to control horizontal cell offset, vertical cell offset and pixel offset within a cell. The image size information such as the number of horizontal cells, vertical cells, offset to current scale, and sub-scale are stored in ROMs. These offsets are added together with image base address to form the actual pixel address. A counter is used to keep track of current sub-scale number and incremented when all addresses in that scale are generated to control the output of
ROMs. As a result, no DSP slices are needed in input/output controllers. The input controller for the classification system operates similarly, but generates three addresses in parallel for three rows of detection windows as discussed in [V-B]. Besides counting the number of scales processed, the output controllers also count the number of histograms/final scores processed for all scales to determine the ending-point of a frame. Each HOG-Engine has a dedicated memory channel for histogram output but three HOG-Engines on one FPGA share a single memory channel for final score output through time multiplexing. Since the number of output values in the final classification are 52x less than the input pixels, multiplexing three outputs into one memory channel will not affect the system throughput. Synchronization between the histogram generation system and classification system are done by a simple 1-bit FIFO. When the histogram output controller finished one scale, it writes one 1-bit value into the FIFO to indicate data available for classification. The classification input controller will read one value out when finished a scale to prevent the FIFO full. The histogram input controller will stop working when the 1-bit FIFO is full (all memory allocated for histograms are used).

To allow maximum throughput for the FPGA execution, we pipelined the HOG-Engine execution at multiple scales/frames. Specifically, after finished fetching pixels at one size, input controller modules will immediately start the next scale/frame, if available. The HOG-Engine operates without knowing the size of the image. However, the histogram generation module needs to know the beginning and ending of each column and each row to combine the cell histograms to block histogram and discard unused values as noted in Section [V-B]. What’s more, since our HOG-Engine operates on two row of cells, the last row will have only one module working if the image has odd number of cell rows. Therefore, the histogram input controller generates a four-bit position signal associated with each pixel to let the core know which portion of image it’s currently executing on. Two-bits indicate the beginning, middle and end of a column and the other two bits used for row. The same idea is also applied to the classification module as the first four columns and last columns will not be sent to all five SVM classifiers. By changing the ROMs containing the image size information and the constant scale number in the input and output controllers, our FPGA implementation can be used for any image sizes and scale factors. As a result, this design is highly scalable.

D. FPGA Resource Usage Comparison

In this section we report the area utilization and clock frequency of the HOG-Engine. The data is shown in Figure 6 for fixed-point, 27 to 13 bits, and single-precision floating-point. The resources usage does not include input and output controllers that interface with external memory. FPGA resources for our actual implementation including all functional units will be discussed later. Percentage values are based on the Xilinx Virtex-6 LX760 FPGA. Compared to floating-point, the registers used for fix-13 are reduced by a factor of 3.0x, LUTs by 6.6x, DSPs by 2.6, BRAMs by 2.2, and frequency increased by 3.1x. The floating-point implementation is fully pipelined with maximum number of pipeline stages applied to most arithmetic operations (generated by Xilinx Core Generator), except the adder at the histogram accumulation. Since the design is fully pipelined, a new vote will be accumulated to the existing bin value every clock cycle. Therefore, a multiple staged floating-point adder can not be used, hence negatively impacts the clock frequency in the floating-point implementation.

VI. RESULTS AND DISCUSSIONS

In this section we evaluate our FPGA program performance and provide a comparison between CPU, GPU, and FPGA. All our tests are based on 640 × 480 images with a scale factor of 1.05. To the best of our knowledge, this is the first densely scanned detection window implementation of HOG algorithm on FPGA. The window stride is four pixels for both directions. Therefore, for each frame, there are 134 scales with window stride of eight as noted in Section [V-B].

A. FPGA Execution Speedup

As discussed in Section [V-A] our FPGA implementation is targeted on Convey HC-2ex computer. Each FPGA is running at 150MHz clock with 16 64-bit memory channels at the same frequency. Three instances of HOG-Engine are implemented on a single FPGA taking all 16 memory channels. The complete system resource utilization, including three HOG-Engine, input/output controllers, and Convey wrapper, is shown in Table V.

As the program is fully pipelined across different image scales, the total execution speed is determined by the number of memory accesses. We performed our experiments on the Convey HC-2ex computer using a single FPGA by measuring only the FPGA execution time. Memory copy time is not taken into account since this latency can be hidden by pipelined
TABLE V: HOG-Engine complete system resource utilization. Three HOG-Engines are instantiated on a single FPGA using all 16 memory channels.

<table>
<thead>
<tr>
<th>Resources</th>
<th>Registers</th>
<th>LUTs</th>
<th>36Kb BRAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage</td>
<td>22%</td>
<td>39%</td>
<td>53%</td>
<td>22%</td>
</tr>
</tbody>
</table>

TABLE VI: HOG detection throughput estimation for different sized images. The throughput for image sizes other than 480 × 640 are estimated based on the number of read requests in the histogram generation module.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>scales</th>
<th>read requests</th>
<th>Input Size (MB)</th>
<th>det. wind</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>640 × 480</td>
<td>34</td>
<td>6219520</td>
<td>12.6</td>
<td>212710</td>
<td>68.18</td>
</tr>
<tr>
<td>800 × 600</td>
<td>38</td>
<td>9900941</td>
<td>20.0</td>
<td>217188</td>
<td>42.99</td>
</tr>
<tr>
<td>1024 × 768</td>
<td>43</td>
<td>10742778</td>
<td>33.7</td>
<td>409186</td>
<td>25.33</td>
</tr>
<tr>
<td>1280 × 960</td>
<td>48</td>
<td>2591328</td>
<td>52.1</td>
<td>637332</td>
<td>16.36</td>
</tr>
<tr>
<td>1600 × 1200</td>
<td>52</td>
<td>40731520</td>
<td>81.8</td>
<td>1049886</td>
<td>10.41</td>
</tr>
</tbody>
</table>

 execution. The experiment with single HOG-Engine indicates the FPGA can process one image at all 34 scales in 44 ms. With three HOG-Engines executing in parallel, we are able to achieve an overall throughput of 68.2 fps on a single FPGA.

For floating-point implementation, only one engine can be placed on a single FPGA with a reduced clock frequency and requires eight input memory channels and four output memory channels for each HOG-Engine. Thus, we estimate its speed of 8.79 fps if under full memory bandwidth (see Table VII, FPGA-fp). Hence, our fixed-point implementation has increased the throughput of the FPGA execution by at least 7.8x. Moreover, we project the speed of executing the fixed-point HOG-Engine on all four FPGAs is 273 fps. In our design, the magnitude and orientation array size of a single image (34 scales) is 12.6 MB, and the size of FPGA output array (final scores) for an image is 0.24 MB. Running at 273 fps requires 3.5 GB/s memory bandwidth which is well below the bandwidth of 15.75 GB/s delivered by the 16x PCIe 3.0.

Based on our single FPGA execution speed, we also estimated the speed for larger images by scaling the throughput based on the number of memory accesses, as shown in Table VI. The number of memory read requests are the input requests for the histogram generation module. Since the design is fully pipelined, the throughput is determined by the memory bandwidth. This estimation can correctly predict the execution speed for different image sizes. As seen in Table VI, the throughput increases by 1.2 times, the number of read requests for the histogram generation grows by 1.6 times. This significant growth is because more image scales are needed to evaluate a single frame.

B. Speedup Comparison

To compare our FPGA implementation with other platforms, we performed HOG pedestrian detection on CPU and GPU. The CPU and GPU implementations are all in single-precision floating-point, adapted from the commonly used OpenCV library [9] to use the parameters that matches the FPGA execution. CPU program is implemented in C++ and compiled by G++ 4.3.6. The CPU platform has two Intel Xeon E5520 quad cores with 24 GB memory. The GPU used is the Nvidia Tesla K20 GPU attached to the same machine. We also include the results of using Intel’s IPP library for CPU’s multi-threading capability. All execution time are measured corresponding to the portions that are implemented on FPGA. In addition, for GPU execution, the memory transfer time is not included. The throughput for all platforms is shown in Table VII. The single FPGA version achieves a 68.7x speedup compared to the single core CPU and a 5.1x speedup compared to GPU. If all four FPGAs are used for execution, we can further push the throughput to 273 fps with a 20x speedup to GPU. As a result, our proposed HOG frame work is suitable for applications that require large throughput and high accuracy pedestrian detection.

C. Power Consumption Comparison

The power consumption estimation for the three platforms is shown in Table VIII. We used the maximum Thermal Design Power of Intel Xeon E5520 processor for the CPU. For the GPU we have used the Nvidia Tesla K20 board power since no individual chip power is available. FPGA power consumption, both fixed-point and floating-point, are estimated using Xilinx Power Estimator 14.3 with a 100% toggle rate (assume all signals will flip every clock cycle). The floating-point module has less power than the fixed-point version. This is due to reduced clock frequency and less resources, since floating-point version only has single-engine running at significantly lower frequency. We compute the power divided by throughput (energy consumption to process a single frame, Joules/Frame) as a measure of power efficiency. Our fixed-point implementation uses 130x less energy than CPU and 31x less energy than GPU to process a single frame.

TABLE VII: HOG detection throughput comparison.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Throughput (fps)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>CPU-IPP</td>
<td>1.14</td>
<td>1.15</td>
</tr>
<tr>
<td>FPGA-fp</td>
<td>8.79</td>
<td>8.86</td>
</tr>
<tr>
<td>GPU</td>
<td>13.40</td>
<td>13.50</td>
</tr>
<tr>
<td>one FPGA-fix13</td>
<td>68.18</td>
<td>68.69</td>
</tr>
<tr>
<td>four FPGA-fix13</td>
<td>272.73</td>
<td>274.77</td>
</tr>
</tbody>
</table>

TABLE VIII: HOG power consumption comparison.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Frame/s</th>
<th>Power (W)</th>
<th>Joules/frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU-IPP</td>
<td>1.14</td>
<td>80</td>
<td>70</td>
</tr>
<tr>
<td>FPGA-fp</td>
<td>8.79</td>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td>GPU</td>
<td>13.40</td>
<td>225</td>
<td>11</td>
</tr>
<tr>
<td>FPGA-fx13</td>
<td>68.18</td>
<td>37</td>
<td>6.54</td>
</tr>
</tbody>
</table>

VII. CONCLUSION

Object and person detections are computationally intensive applications whose importance has been steadily growing. The Histogram of Oriented Gradients (HOG), one of the most popular detection algorithms, achieves a high detection accuracy but delivers just under one frame-per-second (fps) on a high-end CPU. All current fixed-point FPGA implementations
use large bit-width to maintain detection accuracy, or perform poorly with reduced data precision. In this paper we explore the FPGA implementation of HOG using reduced bit-width fixed-point representation to lessen the required area resources on the FPGA, increase the clock frequency and hence the throughput per device. We evaluate the detection accuracy of the fixed-point HOG by the state-of-the-art computer vision pedestrian detection evaluation metrics and show it performs as well as the original floating-point code from OpenCV. We then show our implementation achieves a 68.7x higher throughput than a high-end CPU, 5.1x higher than a high-end GPU, and 7.8x higher than the same implementation using floating-point on the same FPGA. Power consumption estimation shows that FPGA uses 130x less energy than CPU and 31x less than GPU to process a single image. The future work involves performance comparison of different detection classifiers under reduced bit-width using the same HOG feature.

REFERENCES


Walid A. Najjar (F’07) is a Professor in the Department of Computer Science and Engineering at the University of California Riverside. His areas of research include computer architectures and compilers for parallel and high-performance computing, embedded systems, FPGA-based code acceleration and reconfigurable computing.

Walid received a B.E. in Electrical Engineering from the American University of Beirut in 1979, and the M.S. and Ph.D. in Computer Engineering from the University of Southern California in 1985 and 1988 respectively. From 1989 to 2000 he was on the faculty of the Department of Computer Science at Colorado State University, before that he was with the USC-Information Sciences Institute. He was elected Fellow of the IEEE and the AAAS.

Amit K. Roy-Chowdhury (SM’09) received the Masters degree in systems science and automation from the Indian Institute of Science, Bangalore, India, and the Ph.D. degree in Electrical Engineering from the University of Maryland, College Park. He is a Professor of Electrical and Computer Engineering at the University of California, Riverside. His research interests include image processing and analysis, computer vision, pattern recognition, and statistical signal processing. His current research projects include vision networks, distributed visual analysis, wide-area scene understanding, visual recognition and search, video-based biometrics (face and gait), and biological video analysis. He has authored the monograph Camera Networks: The Acquisition and Analysis of Videos over Wide Areas, and published about 150 papers and book chapters. He has been on the organizing and program committees of multiple conferences and serves on the editorial boards of a number of journals.