Heterogeneous Acceleration of HAR Applications

Jose Rodriguez Borbon, *Student Member, IEEE,* Xiaoyin Ma, *Member, IEEE,* Amit K. Roy-Chowdhury, *Senior Member, IEEE,* and Walid Najjar, *Fellow, IEEE*

***Abstract*—Human action recognition (HAR) is an important field of research that intercepts with areas such as image processing, computer vision, and the design of fast algorithms, among others. HAR has several important applications including healthcare monitoring, security and surveillance, assisted living, smart homes, and video search and indexing. Despite recent developments in the field, major challenges remain. For instance, HAR is computationally expensive. Tasks such as video preprocessing, feature extraction, feature quantization, and feature classification require the execution of millions of arithmetic operations for a video sequence lasting a few seconds. To address these problems, we propose a *heterogeneous* approach that is based on an extensive algorithmic and experimental analysis of the histogram of gradients (HOG3D) application. We divide the application into four stages and evaluate each on CPU, GPU, and FPGA platforms. Our heterogeneous design combines the strengths of both FPGA and GPU platforms, and achieves a $1.3X$ speedup compared with a state-of-the-art GPU while being $1.5X$ more energy efficient than other homogeneous solutions, including FPGA-based designs. Moreover, our heterogeneous HAR design using fixed-point arithmetic has comparable accuracy to those of HAR algorithms using single precision floating point arithmetic.**

***Index Terms*—HAR, HOG3D, Accelerators, FPGAs, GPUs.**

I. Introduction

Human action recognition (HAR) algorithms take one or more video sequences as input, usually a few hundred frames, and produce one or more output(s) categorizing the possible action(s) executed by the actor(s) within the video clip(s). Applications of HAR algorithms include health care, assisted living, surveillance, automated video indexing, security, autonomous navigation, robotics, mobile computing, etc. Even though significant progress has recently been made in the design and implementation of HAR applications, several challenges remain: higher throughput for handling large video sequences, lower complexity for real-time applications, highly parallel implementations for faster response times, and energy efficient designs for embedded and mobile applications [1], [2], [3].

HAR algorithms rely on the extraction of video features. These can be computed at regular positions (called dense sampling) or at points of interest (sparse sampling). Video features can be designed by experts in the field, called hand-crafted features (HCF), such as histogram of gradients (HOG), or they can be inferred using machine learning techniques or learned features, such as convolutional neural networks (CNNs). Once the video features are extracted, they can be used to train a classifier, such as a support vector machine (SVM) or a softmax classifier.

HAR implementations based on CNNs have been shown to achieve a higher recognition accuracy than HCF HAR algorithms. However, this advantage comes at a price: lower throughput, higher computational load and costly energy consumption per frame. *Suleiman et al.* [4] shows that HCF HAR algorithms are $311X$ more energy efficient than their CNN counterparts. When the features are learned with larger CNNs, the throughput gap grows to the order of the thousands. Moreover, *Zou et al.* [5] shows that HAR algorithms based on learned features with only three convolutional layers have comparable accuracy and $100X$ higher energy usage than HCF HAR algorithms. As the accuracy of the CNN increases, the energy gap grows dramatically.

The proliferation of video cameras and other forms of image sensing technologies have pushed a large part of the video processing tasks to the edge devices and hence have increased the pressure on achieving high processing rates at low energy budgets. *Our objective is the explore and evaluate the designs of HOG3D-based HAR that can achieve both high throughput and low energy consumption while maintaining acceptable levels of accuracy.*

In this paper, we extend the work presented in [6], whose focus was the performance evaluation of HOG3D HAR algorithm [7] on FPGAs, by evaluating the performance and energy consumption of HOG3D implementations on FPGAs, GPUs, and CPUs. We have profiled the performance of the different HOG3D stages, namely preprocessing, cell descriptor computation, block descriptor computation and video descriptor computation. Based on this analysis, along with the supporting experimental data, we have identified the strengths and weaknesses of each accelerator for HOG3D. By combining the strengths of each platform, we propose a high performance heterogeneous implementation that takes advantage of the strengths of both FPGAs and GPUs, thereby achieving a higher throughput as well as a lower energy consumption per frame than either homogeneous implementation.

For the FPGA, we have implemented the HOG3D application on the Micron Wolverine 2000 with a Xilinx Virtex 7 FPGA and 32 GB of local memory [8]. For the GPU, we have implemented the design on the NVIDIA $K20$, $K40$ and $K80$ [9]. These implementations are compared to a multi-threaded software application running on the Intel Xeon-E5520 quad-core CPU.

Our contributions in [6] includes (1) a comprehensive evaluation of HOG3D algorithm using reduced bit-size fixed-point arithmetic. This reduced bit-size fixed-point algorithm has

J. M. Rodriguez and W. A. Najjar are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521 USA (e-mail: jrodr050@ucr.edu, najjar@cs.ucr.edu)

X. Ma and A. K. Roy-Chowdhury are with the Department of Electrical and Computer Engineering, University of California at Riverside, Riverside, CA 92521 USA (e-mail: xma003@ucr.edu; amitrc@ece.ucr.edu)

Manuscript received Month 05 2018; revised Month 09, 2018.

comparable accuracy to that of HAR algorithms implemented in single or double-precision floating-point arithmetic, and (2) A high-throughput GPU implementation of the HOG3D algorithm that achieves $166.8X$ speedup over the CPU one as well as $3.1X$ speedup when compared with the FPGA design. The FPGA implementation achieves a $53.8X$ speedup over the CPU. Furthermore, while the energy efficiency of the CPU implementation is well below one frame/joule, the GPU design energy efficiency is 5.4 frames/joule.

The new contributions in this work are:

- A detailed I/O and computational complexity analysis for each of the four modules we have identified in our HOG3D design. Based on this analysis, along with our experimental measurements of the throughput and energy consumption per platform, we have identified the strengths and weaknesses of both FPGAs and GPUs accelerators.
- We propose and evaluate a heterogeneous design that seamlessly combines both FPGA and GPU platforms in a single system: the video pre-processing is executed on the FPGA and the video descriptor extraction is executed in the GPU. This heterogeneous design demonstrates a $1.3X$ speedup over the GPU and is $1.5X$ more energy efficient than either homogeneous designs when applied on VGA data as opposed to QVGA data as in [6]

The remainder of this paper is organized as follows: In Section 2, we cover the relevant background. In Section 3, we formally introduce the HAR classification problem. In Section 4, we describe the accuracy of the proposed fixed-point HOG3D algorithm. In Sections 5, 6, and 7, we describe our HOG3D design in FPGAs, GPUs, as well as the computational complexity of the designs. In Section 8, we analyze the throughput and the energy efficiency results for FPGAs and GPUs. Additionally, we present a high throughput energy efficient heterogeneous HOG3D design. Finally, we state our conclusions and future work is devised.

II. Related Work

Early HAR applications are based on HCF and consist of four steps: sampling the video signal, computing features per region of interest, merging these features to get a fixed-size video feature, and finally, training a classifier. Sampling is dense or sparse [1]. Techniques to compute the features of a region include the scale-invariant feature transform (SIFT) [10] and the histogram of oriented gradients (HOG) [11], [7]. The features of the regions are usually merged via a bag-or-words approach [12], [13], and SVMs are commonly used for classification [14]. Initial work in this field includes a behavioral recognition system via sparse spatio-temporal features [15]. Similarly, spatio-temporal features, along with local SVMs, have been proposed [14].

Recent approaches to HAR algorithms are based on learned features [16]. In this approach, a machine learning algorithm samples the video at predetermined positions, learns the local features, aggregates these features, and finally, classifies them. Early work using learned features includes a biologically-inspired system for action recognition [17]. This system takes inspiration from the dual stream organization of the visual cortex: one stream processes the shapes while the second stream processes the motion. Also, a CNN containing a three-dimensional receptive field learns to classify human actions [18]. This network generates action descriptions and uses a feed-forward NN in the classification stage.

In order to improve the accuracy of traditional CNNs, one stream CNNs, researchers have studied two-streams CNNs [19], [20], [21], [22], [23], [24]. In a typical configuration, the first stream learns the spatial features while the second stream learns the temporal features. Variations on this model set one of the streams to learn the features of the optical flow, the motion flow, or the context of the scene, among others. Moreover, the outputs of the streams are usually fused via a fully connected feed-forward neural network. Further, to reduce the computational complexity of two-streams CNNs, factorized CNNs are proposed [25]. Factorized CNNs using spatial convolutional kernels along with temporal convolutional kernels are designed to reduce the complexity of the CNNs while maintaining the recognition accuracy.

Hybrid methods using HCF and learned features have been studied as well. In this approach, the fusion of HCFs boost the performance of the CNNs. Likewise, the fusion of learned features boost the performance of HCF-based classifiers. These designs include a method for recognizing human actions via the fusion of HCF features, based on dense trajectories, and deep-learned features [26]. Also a system for human detection and tracking that uses learned features and SVM classifiers [27]. Further, to save computations, it has been evaluated whether features extracted from CNNs can be repurposed for related tasks [28].

Hou et al. [29] proposes an FPGA real-time HAR system operating at 600 fps. It has a recognition rate of 93.2% when working with a human gesture database with four actions. The recognition rate drops to 80.8% when a few additional gestures are added. Although this system has a competitive throughput, its recognition rate is nontrivial to predict when working with challenging benchmarks having a larger number of classes. Conversely, our system achieves *competitive accuracy* with benchmarks having over 50 classes. Additionally, our design has a *larger throughput* ranging from 455 fps to $1,304$ fps.

III. Problem Description

The four stages of the HOG3D algorithm are as follows[1]:

(a) Preprocessing: In this step, the algorithm computes the partial derivatives along the x, y, and t axes

$$\begin{aligned} dx &= p[x+1,y,t] - p[x,y,t] \\ dy &= p[x,y+1,t] - p[x,y,t] \\ dt &= p[x,y,t+1] - p[x,y,t] \end{aligned} \tag{1}$$

Next, the algorithm computes the integral of the derivatives

$$v_{\partial x}[x,y,t] = \sum_{y' \leq y} \sum_{x' \leq x} dx[x',y',t] \tag{2}$$

[1]In this work, the terms *features* and *descriptors* are used interchangeably.

$v_{\partial y}[x, y, t]$ and $v_{\partial t}[x, y, t]$ are computed in a similar fashion. Finally, the routine computes the integrals videos

$$iv_{\partial x}[x, y, t] = \sum_{t' \leq t} v_{\partial x}[x, y, t'] \quad (3)$$

$iv_{\partial y}[x, y, t]$ and $iv_{\partial t}[x, y, t]$ are computed similarly.

(b) Cell Descriptor Computation: HOG3D considers the set of integral videos as a spatiotemporal volume. Volumes are sampled using a 3D block. Blocks are further divided into $r \times r \times r$ cells. In addition, cells are divided into $s \times s \times s$ sub-blocks. For each sub-block, the algorithm computes the mean gradient vector $\bar{\boldsymbol{g}}_b = [\bar{g}_{b\partial x}, \bar{g}_{b\partial y}, \bar{g}_{b\partial t}]^T$. The component $\bar{g}_{b\partial x}$ is computed as

$$\bar{g}_{b\partial x} = J(t + l) - J(t) \quad (4)$$

where $J(t) = iv_{\partial x}[x, y, t] + iv_{\partial x}[x + w, y + h, t] - iv_{\partial x}[x, y + h, t] - iv_{\partial x}[x + w, y, t]$. Here, w, h and l are implementation parameters. Similar equations are used to compute $\bar{g}_{b\partial y}$ and $\bar{g}_{b\partial t}$. Subsequently, the algorithm quantizes each vector $\bar{\boldsymbol{g}}_b$ using a regular icosahedron. To quantize $\bar{\boldsymbol{g}}_b$, the routine centers the icosahedron at its origin in a three dimensional space. Let $P_{k,3}$ be the matrix where each row contains the icosahedron coordinates of the central point of face i

$$P_{k \times 3} = \begin{bmatrix} p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & p_{22} \\ \ldots & \ldots & \ldots\ldots \\ p_{k0} & p_{k1} & p_{k2} \end{bmatrix}$$

HOG3D calculates the normalized quantization vector $\hat{\boldsymbol{g}}_b$ by computing

$$\hat{\boldsymbol{g}}_b = \frac{P \times \bar{\boldsymbol{g}}_b}{||\bar{\boldsymbol{g}}_b||_2} \quad (5)$$

Next, the algorithm thresholds the elements of vector $\hat{\boldsymbol{g}}_b$ using a given parameter α_1. If $\hat{\boldsymbol{g}}'_b$ is the resulting vector after the threshold operation (if $\hat{g}_b[j] < \alpha_1$ then $\hat{g'}_b[j] = 0$ else $\hat{g'}_b[j] = \alpha_1 - \hat{g}_b[j]$) then, the routine uses a scaling factor to obtain the sub-block descriptor

$$\boldsymbol{g}_b = \frac{||\bar{\boldsymbol{g}}_b||_2}{||\hat{\boldsymbol{g}}'_b||_2} \hat{\boldsymbol{q}}'_b \quad (6)$$

Then, HOG3D computes the vector $\boldsymbol{c}'$ by adding, element by element, the $s \times s \times s$ sub-block descriptors inside the cell

$$c'[j] = \sum_{i=0}^{s \times s \times s - 1} g_{bi}[j] \quad j = 0, .., k - 1 \quad (7)$$

In addition, the routine normalizes $\boldsymbol{c}'$. The resulting vector is the cell descriptor

$$\boldsymbol{c} = \frac{\boldsymbol{c}'}{||\boldsymbol{c}'||_2} \quad (8)$$

(c) Block Descriptor Computation: HOG3D calculates the block descriptor $\boldsymbol{h}$ by concatenating the cell descriptors inside the block

$$\boldsymbol{h} = \{\boldsymbol{c}_{r \times r \times r - 1}, .., \boldsymbol{c}_1, \boldsymbol{c}_0\} \quad (9)$$

Here $\boldsymbol{h} \in R^d$ a d-dimensional space. The result of this step is a set of block descriptors $H = \{\boldsymbol{h}_i\}_{i=0..n-1}$.

(d) Video Descriptor Computation: Because the number of descriptors changes from video to video, a technique for aggregating varying size descriptors into a fixed-size descriptor has to be implemented [13]. In here, a vocabulary $D = \{\boldsymbol{d}_j\}_{j=0..m-1}$ with $\boldsymbol{d}_j \in R^d$ is given. To compute fixed-size descriptors, HOG3D computes the distances between each block descriptor $\boldsymbol{h}_i$ and each visual word $\boldsymbol{d}_j$. Next, the algorithm increments by one the histogram slot of the visual word $\boldsymbol{d}_j$, i.e. $x[j]$, that is closest to $\boldsymbol{h}_i$. The resulting histogram $\boldsymbol{x} \in R^m$ is used as the video descriptor. Finally, the routine uses the video descriptor $\boldsymbol{x}$ as input of a classifier. Notice that $J(t + l) - J(t)$ is the sum of the pixels between t and $t + l$, excluding t, in the area of rectangle (x, y, w, h). In our design, l is always two, and as a result, our design gets simplified. First, the integral video images are computed between adjacent integral images only

$$iv_{\partial x}[x, y, t] = v_{\partial x}[x, y, t'] + v_{\partial x}[x, y, t' - 1] \quad (10)$$

Second, because the computation of $J(t + l) - J(t)$ excludes t, $\bar{g}_{b\partial x}$ is computed as

$$\bar{g}_{b\partial x} = J(t + l) \quad (11)$$

Further, $\bar{g}_{b\partial y}$ and $\bar{g}_{b\partial t}$ are computed similarly.

As shown above, the implementation of the HOG3D algorithm requires the normalization of a number of low dimensional vectors, see (5), (6), and (8). As a result, the Euclidean norm has to be computed. To optimize hardware resources, the Euclidean norm can be approximated as proposed in [30]

$$\|\mathbf{u}\|_2 \approx (1 - \lambda) Max(|u[i]|_{i=0,..,p-1}) + \lambda \sum_{i=0}^{p-1} |u[i]| \quad (12)$$

with $\lambda < 1$. As shown in this equation, this method is inexpensive to implement in hardware as it does not require the implementation of the resource-hungry square root operation.

IV. Fixed-Point HOG3D HAR

In this section, we report on the evaluation of the fixed-point HOG3D recognition accuracy using four benchmarks:

- The KTH benchmark is a collection of 599 videos with six actions [14].
- The UCF11 benchmark is a collection of 1,600 videos with 11 action categories including basketball shooting, horseback riding, swinging among others [31], [32].
- The UCF50 benchmark is a collection of 6,680 videos with 50 actions [33]. This benchmark includes all the actions in UCF11, plus 39 additional actions. As with UCF11, this dataset is challenging due to its diverse conditions as well as the number of actions.
- The UCF101 benchmark [34], a collection of 101 human actions containing 13,320 video clips. This benchmark extends the UCF50 by adding additional actions. This benchmark is particularly challenging due to the diverse set of conditions including illumination, viewpoint, scale, camera motion, backgrounds, etc.

Our fixed-point HOG3D implementation is based in the double-precision floating-point implementation described in [7]. Starting from this source code, we added dense sampling, fixed-point arithmetic, and half-precision floating arithmetic. To sample the input video, our routine uses a 3D block. The overlapping between adjacent blocks is 50%. While our routine keeps the temporal scale fix, the spatial scale is increased by a factor of $\sqrt{2}$ until the size of the block is larger than the size of the image. Our algorithm divides each 3D block into 64 cells, four cells per dimension. Furthermore it divides each 3D cell into eight sub-blocks, two sub-blocks per dimension. As a result, the size of the block descriptor is 640 elements: $64 = 4 \times 4 \times 4$ cell descriptors and ten elements per cell descriptor when the algorithm uses half of the icosahedron orientations. For each of the HOG3D stages, we set the input bit-width as well as the output bit-width; if m is the number of integer bits and n is the number of fractional bits, the total bit-width is $m + n$. To minimize overflows and underflows, the operands have been normalized whenever possible. The maximum bit-width is set to 27 bits and the minimum to eight bits. For further details refer to our previous work [6].

To evaluate the accuracy of our HAR recognition method, our algorithm uses reduced fixed-point arithmetic along with a modified version of the SVM library LIBSVM [35]. Here, we added a χ^2 kernel. In addition, our algorithm observes the experimental settings described in [7]. In particular, we use leave-one-group-out cross validation. Since the videos in every dataset are grouped, said N groups, we train a SVM with $N - 1$ groups and make predictions about the videos in the left-out-group. If the left-out-group has k videos and p predictions are correct, the recognition accuracy is p/k. We repeat this process for all the groups and report the average recognition accuracy for both floating point and fixed-point precision.

The results are shown in Fig. 1. The $'half'$ and $'single'$ results are from our modified HOG3D implementation working in half and single precision floating point. The $'fxp27'$ down to $'fxp8'$ results correspond to the fixed-point HOG3D implementation when working with 27 down to eight bits. For the UCF101 benchmark, we only report the recognition accuracy for single precision floating point and for $'fxp27'$, $'fxp16'$, and $'fxp8'$ fixed-point precision.

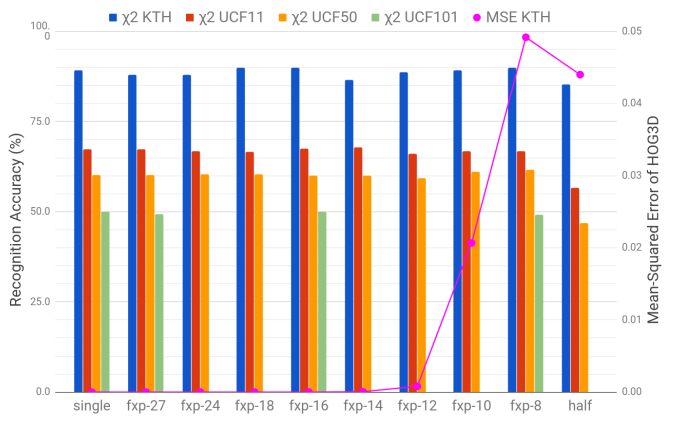


Fig. 1. On the left, the reduced fixed-point recognition accuracy using χ^2 kernel versus bit-width. On the right, the mean-squared error (MSE) of the video descriptors for the KTH dataset.

The accuracy of the original double-precision floating point implementation [7], and our $'single'$ precision floating point implementation are comparable for all the benchmarks [6]. Moreover, the recognition accuracy for the KTH benchmark is high, it decreases for the UCF101 benchmark for all fixed-point precisions. This is consistent with the fact that the UCF101 is the hardest benchmark to recognize. The recognition accuracy behavior is significant for reduced fixed-point arithmetic. As show in the figure, as the bit-width decreases from 27 bits to eight bits, the recognition accuracy is comparable to that of the single precision floating point albeit small fluctuations. The half-precision implementation has the lowest overall recognition accuracy. This behavior is mostly due to the characteristics of the range and the precision of half-precision floating point numbers. In the case of reduced fixed-point arithmetic, the range and precision are dynamic; they change from stage to stage while the range and precision of the half-precision floats remain static [6].

Moreover, we compute the mean-squared error (MSE) by comparing the values of the fixed-point video descriptor with those of the double-precision video descriptors, the ground truth. In Fig. 1, we only report the MSE of the KTH dataset because it has the largest value. As shown in the plot, the MSE is well below 1×10^{-2} for twelve bits and above. For ten bits and eight bits, the MSE increases, although it always remains below 5×10^{-2}. In brief, these results show it is feasible to implement HOG3D in reduced fixed-point arithmetic without compromising its accuracy.

V. FPGA Implementation

In this section, we describe the implementation of HOG3D in FPGAs. In this design, all arithmetic operations use reduced fixed-point operands. Operations such as multiplications and divisions have been implemented in Xilinx fixed-point cores [36]. When the result of an arithmetic operation overflows the result is saturated on-the-fly. Due to the design of the DSP units in the Virtex-7 FPGAs and to minimize logic usage, the result of operations including multiplications and divisions are always truncated [37].

The input of the algorithm are streams of gray-scale videos consisting of 97 images. The output is the video descriptor vector $\boldsymbol{x}$ with 1000 elements. Unless otherwise described, four videos are moved from the CPU to the FPGA off-chip memory for processing. Then four engines process each video. The description of each engine is given in what follows.

A. Preprocessing Engine

This engine is responsible for computing the integral videos. Fig. 2 shows the modules responsible for computing the integral videos along the x, y, and t axes. In this design, all communications between modules are implemented via FIFOs [38]. As shown in Fig. 2, the computation of the integral videos is straightforward. The *read image* module reads three images at a time from off-chip memory, the images at indexes t, $t+1$, and $t+2$, in a row by row fashion. Next, the *gradients* module computes the derivatives of the input pixels along the x, y, and t axes. The *integral image* module computes the

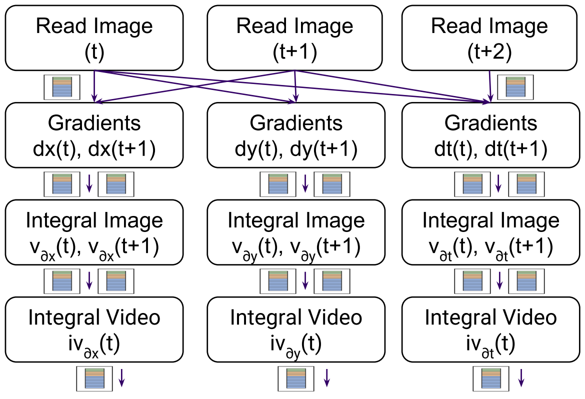

Read Image (t)
Read Image (t+1)
Read Image (t+2)
Gradients dx(t), dx(t+1)
Gradients dy(t), dy(t+1)
Gradients dt(t), dt(t+1)
Integral Image $v_{\partial x}(t)$, $v_{\partial x}(t+1)$
Integral Image $v_{\partial y}(t)$, $v_{\partial y}(t+1)$
Integral Image $v_{\partial t}(t)$, $v_{\partial t}(t+1)$
Integral Video $iv_{\partial x}(t)$
Integral Video $iv_{\partial y}(t)$
Integral Video $iv_{\partial t}(t)$

Fig. 2. Preprocessing engine: Four modules are responsible for the computation of the integral videos along the x, y, and t axis.

integrals of the gradients. To do so, for each input array, it calculates the integral of the current row in a register. Also, this module maintains an on-chip copy of the integral of the previous row. By adding these two integrals, this module obtains an integral image. The *integral video* module takes as input two integral images per axis, adds them together, and writes the integral video into a FIFO. Finally, the resulting integral videos are written to the off-chip memory.

Notice that in our design, the use of FIFOs facilitates the communication between modules as well as the modularization of the design. Each module reads from inputs FIFOS, execute the required computations, and write results to the output FIFOs. In summary, this design reads 97 gray-scale images per video and outputs $144 = 48 \times 3$ pairwise integral videos with two bytes per element. On-chip computations are performed in reduced fixed-point arithmetic with either 8 or 16 bits operands. To improve throughput, we replicate this engine eight times. As a result, this engine can process eight videos in parallel.

B. Cell Descriptor Engine

Fig. 3 shows the modules responsible for the computation of the cell descriptors. The *read integral videos* module reads

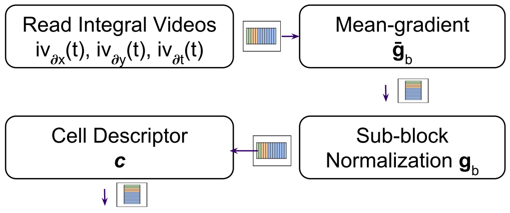

Read Integral Videos $iv_{\partial x}(t)$, $iv_{\partial y}(t)$, $iv_{\partial t}(t)$
Mean-gradient $\bar{\mathbf{g}}_b$
Cell Descriptor $\boldsymbol{c}$
Sub-block Normalization $\mathbf{g}_b$

Fig. 3. Components of the cell descriptor engine.

the integral videos from off-chip memory as required in the computation of the mean gradient vector $\bar{\boldsymbol{g}}_b$. The *mean-gradient* module computes vector $\bar{\boldsymbol{g}}_b$ along with its norm. The *sub-block normalization* module computes the sub-block descriptor $\boldsymbol{g}_b$ by executing the matrix vector multiplication $P \times \bar{\boldsymbol{g}}_b$. Matrix $P_{10\times 3}$ is stored on-chip.

The *cell descriptor* module computes the normalized vector $\boldsymbol{c}_j$. Since the computation of the sub-block descriptors proceeds in a cell by cell order, each time a sub-block descriptor is computed, the unnormalized cell descriptor is updated as described in (7). Next, this module computes the normalized cell descriptor and writes the results into a FIFO. Finally, the normalized cell descriptors are written to the off-chip memory.

To take advantage of the FPGA resources, we replicate this engine four times. Hence, our design processes four videos in parallel. For each incoming integral video and for each cell descriptor, this engine computes two sub-block descriptors in parallel. Furthermore, parallel calculations have been implemented when feasible. In the case of the operation $P \times \bar{\boldsymbol{g}}_b$, thirty multiplications are executed in parallel. In the case of vector normalizations, divisions and multiplications are executed in parallel as well. In brief, for each incoming video, this engine reads the pairwise integral videos, computes the sub-block descriptors, and outputs the normalized cell descriptors vectors. While the inputs to this module are 2-bytes arrays, the outputs are ten-element vectors. On-chip operations are executed in 16 bits. After vector normalizations, the width of the elements in the output vector reduces to eight bits.

C. Block Descriptor Engine

This engine reads the normalized cell descriptors, computes the block descriptors, transposes the block descriptors, and writes the results to the off-chip memory. This engine is composed of three modules, as shown in Fig. 4.

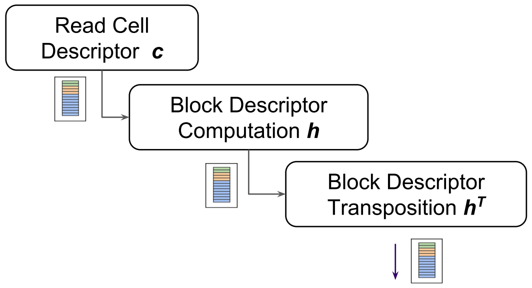

Read Cell Descriptor $\boldsymbol{c}$
Block Descriptor Computation $\boldsymbol{h}$
Block Descriptor Transposition $\boldsymbol{h}^T$

Fig. 4. Components of the block descriptor engine.

The *read cell descriptor* module reads the normalized cell descriptors $\boldsymbol{c}_j$ from the off-ship memory. Next, the *block descriptor* module concatenates sixty-four cell descriptors and writes the resulting vector, $\boldsymbol{h}$ size 64×10, into a FIFO. Then, the *transposition* module transposes the descriptors. In this process, the output of the *block descriptor* module is written into eight FIFOs, with each FIFO containing one block descriptor. Finally, this module pops the eight FIFOs and writes the results of the off-chip memory, one column per FIFO, i.e. the transpose operation.

To gain performance, we replicate this engine four times, and as a result, four videos are processed in parallel. For each incoming video and for each block descriptor, this engine reads and writes four cell descriptors in parallel. In short, the input of this module is an array of cell descriptors and the output is an array of block descriptors. All operands in this module are one-byte wide.

D. Video Descriptor Engine

The next step is to compute the video descriptors. This engine takes two inputs; the first input is the set of block descriptors H, and the second input is a set of pre-computed cluster centers D. The goal of this module is to find for each element in H the nearest neighbor in D, and finally, to find the distribution of the block descriptors per each given center. In this design, the set D is mapped to the reference matrix $R_{m\times d}$, and the set H is mapped to the query matrix $Q_{d\times n}$. As a result, nearest neighbor problem can be formulated as a matrix multiplication problem i.e. $C_{m\times n} = R \times Q$ with $c[i,j] = \sum_{k=0,..,d-1}(r[i,k] - q[k,j])^2$. Thus, matrix C contains all the distances between the given m centers and the n query points.

Matrix multiplications on FPGAs has been studied extensively [39], [40]. In this work, we have followed the directions of the design proposed in [41] with modifications. The computation of matrix C is blocked. For illustration purposes, let us assume that the size of every block C_{ij} is $p \times p$, moreover, that $m = k * p$ and $n = s * p$. Matrix C can be written as

$$C_{m\times n} = \begin{bmatrix} C_{10} & C_{11} & ... & C_{1s} \\ C_{20} & C_{21} & ... & C_{2s} \\ ... & ... & ... & ... \\ C_{k0} & ... & ... & C_{ks} \end{bmatrix}$$

The computation of sub-matrices C_{ij} is from top to bottom and from left to right. In this work, matrix R has $n \times 640$ elements. The reference centers have been normalized off-line. The components of this engine are shown in Fig. 5.

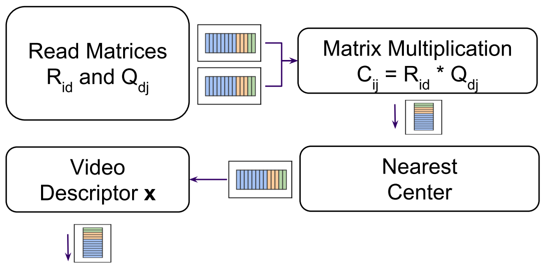


Fig. 5. Components of the video descriptor engine.

The *read matrices* module is responsible for reading the columns of sub-matrix R_{id} one at the time into a FIFO, in addition, the rows of sub-matrix Q_{dj} one at the time into p FIFOs. The *matrix multiplication* module executes the multiplication $C_{ij} = R_{id}Q_{dj}$. The layout of this component is shown in Fig. 6. At the beginning, this module reads the first element in the FIFO containing $r[0,0]$ and it also reads the p FIFOs containing $q[0,0], q[0,1], .., q[0,p-1]$. Next, p subtraction-and-multiplications are executed in parallel. The results are stored in p BRAM accumulators with each accumulator having p addresses and 16 bits per address. This process continues until the last element, $r[p-1,0]$, of the first column in R_{id}

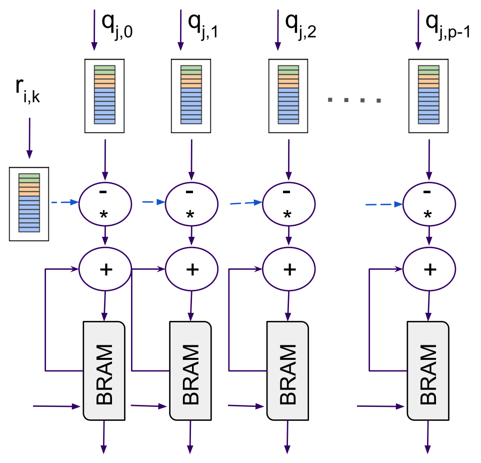


Fig. 6. Matrix multiplication component . The top p FIFOs contain the elements rows of matrix Q. The left-most FIFO contains the elements of the columns of matrix R. In the center, the distances are computed and accumulated.

is multiplied by the current row $q[0,0], q[0,1], .., q[0,p-1]$. Next, this module executes the outer-product between the elements in the second column of R_{id} and the second row of Q_{dj}. This module continues to execute outer-products until the calculation of the sub-matrix C_{ij} is complete. Results are written into p FIFOs. We notice that by incrementing the number of operation executed in parallel, the parameter p, we can take advantage of the DSPs present in the FPGA: the larger is p, the greater the performance.

The *nearest center* module finds the nearest center for every object in Q. Each time a sub-matrix C_{ij} is computed, this module reads from p FIFOs. For every FIFO, i.e. for every object $\boldsymbol{q}_j \in Q$, this module keeps track of the minimal distance and the associated center thus far. Since the sub-matrices C_{ij} are computed top-to-bottom and left-to-right, each time that a bottom sub-block is computed i.e. $C_{k0}, C_{k1}, .., C_{ks}$, this module outputs the centers associated with $\boldsymbol{q}_j$. These centers are written to p FIFOs.

The *video descriptor* module reads the outputs of the previous module. Every time that this module reads a center, the BRAM memory address associated with that center is incremented by one. When all the nearest centers are found, this module outputs the video descriptor vector $\boldsymbol{x}$, to the off-chip memory. Notice that the computation of vector $\boldsymbol{x}$ can potentially harm the throughput as the *nearest center* module outputs as many as 320 centers per cycle. To speed up the computation of $\boldsymbol{x}$, we use a reduction tree with ten nodes at the top level. Each of those node computes local video descriptor by processing 32 inputs. In the next level of the tree, the local video descriptors are merged into pairs. This reduction continues until the final video descriptor is found.

To improve throughput, we replicate this engine four times such that four videos are processed in parallel. For each engine, the parameter p has been steadily increased until the resources in the FPGA are nearly exhausted. In this design, we set p to 320 such that 1,280 multiplications are executed in parallel. While all input elements are one byte, the output elements are two bytes. On-chip computations are executed in two bytes.

VI. GPU Implementation

In this section, we describe the implementation of HOG3D in GPUs. We use 32-bit integer arithmetic and single precision floating point arithmetic. Our implementation processes eight videos in parallel by taking advantage of CUDA streams [9], [42]. In this scenario, each stream is responsible for processing one video. Moreover, kernel calls are issued in a breadth-first fashion across all the running streams. For the purpose of illustration, let us assume that eight CUDA streams $S1$,..,$S8$ are running in parallel and each stream has two kernels $K1$ and $K2$. The GPU executes kernel K_1 on all eight streams: $S1(K_1), .., S8(K_1)$ followed by $S1(K_2), .., S8(K_2)$. In our design, the GPU executes eight streams. For each stream, one video having 97 gray-scale images is transferred from the host main memory to the GPU off-chip memory. Eventually, the HAR algorithm is executed using four engines as described below.

A. Preprocessing Engine

The *preprocessing* engine computes the pairwise integral videos along the x, y, and t axis. Fig. 7 shows the kernels used in this engine. The *image gradients* kernel computes the

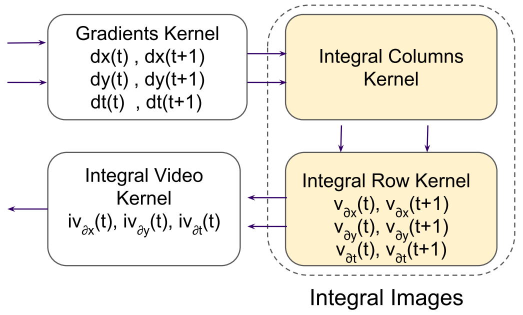

Gradients Kernel
dx(t) , dx(t+1)
dy(t) , dy(t+1)
dt(t) , dt(t+1)
Integral Columns Kernel
Integral Video Kernel
$iv_{\partial x}(t)$, $iv_{\partial y}(t)$, $iv_{\partial t}(t)$
Integral Row Kernel
$v_{\partial x}(t)$, $v_{\partial x}(t+1)$
$v_{\partial y}(t)$, $v_{\partial y}(t+1)$
$v_{\partial t}(t)$, $v_{\partial t}(t+1)$
Integral Images

Fig. 7. Components of the pre-processing engine.

gradients along the x, y, and t axis. In this step, each image is divided in tiles of size 16×16. The partition of an image into tiles facilitates coalesced I/O operations. Each tile is then loaded into shared memory along with the halo elements. For every tile, the kernel computes the gradients along the x, y and t axis.

Subsequently, the *integral images* kernel, the right hand side of Fig. 7, reads the gradients and computes the integrals of the images. Calculating the integral of the images is challenging for GPUs due to the presence of thread divergences [43], [44]. Our design is similar to the work presented in [44], with modifications.

In the first step, the *integral columns* kernel reads the gradients, using one CUDA thread per column, integrates the values of the columns, and writes the resulting integrals to the off-chip memory. While the first step can be executed efficiently by one CUDA thread, the computation of integrals along the rows requires synchronization between the threads in a CUDA block. In the second step, the *integral row* kernel reads the computed column integrals into shared memory. The algorithm computes the row integrals in two phases: the up-sweep phase and the down-sweep phase. In the up-sweep phase, the kernel computes the prefix-sum for all odd elements. In the down-sweep phase, the kernel computes the prefix-sum for all even elements. In our work, because the rows of the input images are no larger than 640, the prefix-sum per row can be implemented in shared memory; the routine sets the row size to $N = 1024$ and pads the data as necessary. After padding, the kernel uses $N/2$ threads, takes $2Log_2(N) - 1$ steps, and executes $2(N - 1)$ additions [44]. The resulting array integrals are then written to the off-chip memory.

Finally, the *integral video* kernel reads two integral images per axis, adds their values, and writes the results back to the off-chip memory. In this kernel, threads are mapped to the columns such that we achieve coalesced memory accesses.

B. Cell Descriptor Engine

In this engine, the algorithm computes the cell descriptors $\boldsymbol{c}_j$. Fig. 8 shows the kernels involved in this computation.

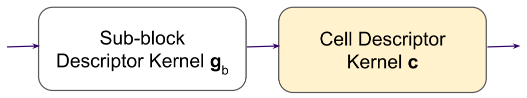

Sub-block Descriptor Kernel $\mathbf{g}_b$
Cell Descriptor Kernel $\mathbf{c}$

Fig. 8. Cell descriptor engine. Two kernels are responsible for the computation of the cell descriptors.

The *sub-block descriptor* kernel computes the mean-gradient vector $\boldsymbol{g}_b$. In this design, a thread is responsible for computing the mean-gradient. To improve the performance, the matrix $P_{10\times3}$ is stored in constant memory and the mean-gradient is stored in shared memory. The *cell descriptor* kernel computes the vector $\boldsymbol{c}$. Specifically, a thread reads the normalized sub-block descriptors inside the cell, adds their values, executes vector normalization, and finally, writes the resulting vector to the off-chip memory. In this design, a thread is responsible for computing the normalized cell descriptors using shared memory.

C. Block Descriptor Engine

The *block descriptor* engine computes the block descriptors using two kernels as shown in Fig. 9. The *block descriptor* kernel reads the cell descriptors $\boldsymbol{c}$ and computes the block

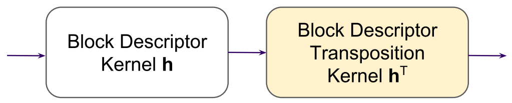

Block Descriptor Kernel $\mathbf{h}$
Block Descriptor Transposition Kernel $\mathbf{h}^T$

Fig. 9. Block descriptor engine. Two kernels are responsible for computing the block descriptors.

descriptor $\boldsymbol{h}$. Results are written to the off-chip memory. Because this kernel is I/O bounded, its performance is improved by increasing the number of threads executing off-chip reads and writes. In this work, a thread is responsible for reading and for writing each element in $\boldsymbol{h}$. The *block transposition* kernel reads, transposes, and writes an array size $640 \times 10,240$. Matrix transposition using tiles is a well-studied kernel [45], [46], [44].

D. Video Descriptor Engine

In this engine, the video descriptor is computed via a nearest neighbor clustering algorithm. Given two vectors $\boldsymbol{x} \in R^d$ and $\boldsymbol{y} \in R^d$, their Euclidean distance is given by

$$\rho(\boldsymbol{x},\boldsymbol{y})^2 = (\boldsymbol{x}-\boldsymbol{y})^T(\boldsymbol{x}-\boldsymbol{y}) = ||\boldsymbol{x}||^2 + ||\boldsymbol{y}||^2 - 2\boldsymbol{x}^T\boldsymbol{y} \quad (13)$$

Furthermore, distances between vectors can be computed via matrices [47]. Let R and Q be two matrices size $d \times m$ and $d \times n$ respectively. R represents m reference centers and Q represents n block descriptors. Let $\rho^2(R,Q)$ be a $m \times n$ matrix containing the distances between the reference centers and the block descriptors. Then $\rho^2(R,Q)$ can be computed as

$$\rho^2(R,Q) = N_R + N_Q - 2R^TQ \quad (14)$$

In this equation, the elements of the j^{th} row of N_R are all equal to $\sum_{i=0}^{i=d-1}(R[i,j])^2$. The elements of the j^{th} column of N_Q are all equal to $\sum_{i=0}^{i=d-1}(Q[i,j])^2$. To save memory, in this design, we represent N_R and N_Q as vectors. Further, because the reference centers are predefined, N_R and R^T are computed off-line. Fig. 10 shows the kernels involved in the computation of the video descriptors.

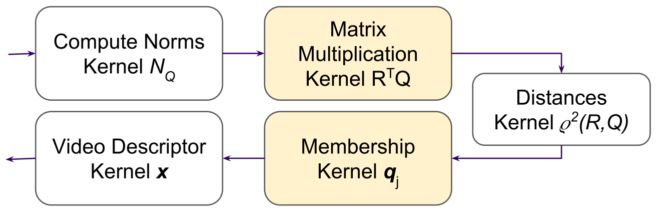

Compute Norms Kernel N_Q
Matrix Multiplication Kernel R^TQ
Distances Kernel $\rho^2(R,Q)$
Membership Kernel $\boldsymbol{q}_j$
Video Descriptor Kernel $\boldsymbol{x}$

Fig. 10. Video descriptor engine. Kernels used in the process of computing the video descriptors.

The *norms* kernel computes vector N_Q. In this design, a CUDA thread is responsible for computing $N_Q[j]$. This assignment makes it possible to optimize off-chip memory bandwidth. The *matrix multiplication* kernel executes R^TQ by means of the CUBLAS library [45]. The *distances* kernel calculates the matrix $\rho^2(R,Q)$. In this work, a CUDA thread is responsible for computing the distances between the reference centers and the block descriptors i.e. $\rho^2(R,Q)[i,j]$. To do so, it reads elements $N_R[i]$ and $N_Q[j]$ along with element $\rho^2(R,Q)[i,j]$. Next, it computes the distance, as shown in (14), and writes the values to off-chip memory.

The *membership* kernel finds the membership of every object in Q. To optimize the bandwidth, a CUDA thread is responsible for computing the membership of $\boldsymbol{q}_j$ by means of scanning column j in matrix $\rho^2(R,Q)$. Results are written to the off-chip membership vector. Finally, the *video descriptor* kernel computes vector $\boldsymbol{x}$ in two steps. In the first step, the privatized step, threads read the elements of the membership vector. As the coalesced reads are executed, the video descriptor is computed, in shared memory, by means of atomic adds. In the second step, the global step, threads write atomically to the shared video descriptor vector in the off-chip memory [48].

VII. Complexity Analysis

In this section, the complexity of the FPGA and the GPU design per engine is given. We assume the HOG3D algorithm takes as inputs 97 gray-scaled images having M rows and N columns. Also, we assume that each sub-block descriptor contains K elements, each cell descriptor contains eight sub-block descriptors, and each block descriptor contains 64 cell descriptors. Moreover, the design processes C cell descriptors, n block descriptors, and m reference centers. Each block descriptor and each reference center vector has $64 \times K$ elements. Moreover, the matrix multiplication $R^T \times Q$ operation is blocked, and the size of the block is p. Without loss of generality, we assume that m/p and n/p are integers.

A. Preprocessing Engine

Table I shows the results of the complexity analysis.

TABLE I
Preprocessing Engine Complexity Analysis

I/O operations	FPGA	$MN(97+144)$
	GPU	$MN(97+144+144\times 8)$
Arithmetic Operations	FPGA	$144(7MN-4M-2N)$
	GPU	$144(9MN-6M-2N)$

In FPGAs, the preprocessing engine requires the reading of 97 images and the writing of 144 integral video images, 48 for each dimension. In GPUs, the preprocessing engine requires $8MN$ additional I/O operations per integral video image: $2MN$ writes due to computation of the gradients, $2MN$ reads and $2MN$ writes due to the computation of the integral of the derivatives, and $2MN$ reads due to the computation of the integral video image.

In FPGAs the computation of the integral videos is executed in three steps. (a) The computation of the derivatives per input image takes $M(N-1)$ operations. (b) The integral of the derivatives, takes $2MN - M - N$ operations. Hence, the computation of the integrals of the gradients for two images requires $2(3MN-2M-N)$ operations. (c) The computation of the pairwise integral videos requires MN additions. In GPUs, steps (a) and (c) are as in the case of FPGAs. Step (b) takes $3MN-2M-N$ operations: the integrals along the columns require $(M-1)N$ operations and integrals along the rows require $2M(N-1)$ operations. Hence, the computation of the integrals of the gradients for two images requires $2(4MN-3M-N)$ operations.

Taking into consideration the complexity analysis and the proposed design for FPGAs and GPUs, we observe that this engine is I/O bounded. In the case of FPGAs, the bound applies despite the larger number of arithmetic operations. This engine computes three integral videos in parallel. In addition, the computation of the gradients, the integral of the gradients, and the pairwise integral videos is pipelined.

B. Cell Descriptor Engine

The complexity analysis is given in Table II.

TABLE II
CELL DESCRIPTOR COMPLEXITY ANALYSIS

I/O operations	FPGA	$C(8(12) + K)$
	GPU	$C(8(12) + 8K + 9K)$
Arithmetic Operations	FPGA	$C(8(11K) + 11K + c)$
	GPU	$C(8(11K) + 11K + c)$

In FPGAs, the computation of one sub-block descriptor takes twelve reads, see (11), and the computation of the cell descriptor takes K writes. In GPUs, the computation of each sub-block descriptor takes K additional writes and the computation of the cell descriptors takes $8K$ additional reads. In FPGAs, as well as GPUs, the computation of the sub-block descriptor $\boldsymbol{q}_b$ takes $11K$ operations[2]. The computation of the cell descriptor $\boldsymbol{c_j}$ requires $11K$ operations[3].

The computation of the cell descriptors in FPGAs is I/O bounded despite the larger number of arithmetic operations. While the arithmetic operations are parallelized and pipelined, at least K operations are executed in parallel per pipeline, the reading of the integral videos is serial. Moreover, in GPUs, the performance of this engine is bounded by the number of arithmetic operations as a CUDA thread is responsible for the computation of each sub-block and cell descriptor.

C. Block Descriptor Engine

In FPGAs, the computation of each block descriptor involves the reading and writing of 64 cell descriptors i.e. $2n(64C) = 2n(d)$ I/O operations. In GPUs, additional $2nd$ I/O operations due to transpositions must be executed. For both FPGAs and GPUs, this engine is I/O bounded.

D. Video Descriptor Engine

Table III shows the results of the complexity analysis.

TABLE III
VIDEO DESCRIPTOR COMPLEXITY ANALYSIS

I/O operations	FPGA	$2(dmn/p) + m$
I/O operations	GPU	$2(dmn/p) + mn$ + $n(d + 2)$ + $2mn$ + $mn + n$
Arithmetic Operations	FPGA	$(3d - 1)(mn) + nm$
Arithmetic Operations	GPU	$(2d - 1)mn$ + $3mn + n(2d - 1)$ + $(m - 1)n + n$

In the FPGA, the computation of each sub-matrix C_{ij} size $p \times p$ takes $2dp$ reads. Moreover, writing the video descriptor vector $\boldsymbol{x}$ takes m writes. Computing each element $c[i, j]$ takes d subtractions, d multiplications, and $d - 1$ additions. As a result, the total number of operations per sub-matrix C_{ij} is $(3d-1)(p^2)$. Finding the membership of each block descriptor takes $(m - 1)$ comparisons. Moreover, the computation of the video descriptor vector takes n additions.

The computation of the video descriptor in GPUs is described in three steps. First, the computation of matrix $R^T Q$ takes $2pd * (mn/p^2)$ reads, mn writes, dmn multiplications, and $(d-1)mn$ additions. Second, the computation of $\rho^2(R, Q)$ is executed in two parts. (a) Computing N_Q requires dn reads, n writes, dn multiplications and $(d - 1)n$ additions. (b) Computing $\rho^2(R, Q)$ requires n reads and the reading and writing of a matrix size mn. Moreover mn multiplications and $2mn$ additions are required. Third, the computation of the video descriptor $\boldsymbol{x}$ requires mn reads, n writes, $(m - 1) * n$ comparisons, and n additions. Finally, this engine is computed bounded.

VIII. RESULTS

In this section, we discuss the throughput and the energy efficiency of the HOG3D design in FPGAs and GPUs. At the end, we propose a heterogeneous HOG3D (HHAR) algorithm.

A. FPGA Synthesis Results

In this part, we describe the results of the synthesis, placing, and routing. The testbed is composed of two Intel Xeon CPUs E5-2640 and two Virtex-7 FPGAs [8]. Each FPGA has 32 memory channels. To achieve maximum bandwidth per channel, 1.25 GB/s, a 64-byte exclusive request has to be issued. Otherwise, a channel can handle request sizes of 1,2,4, or 8 bytes at the expenses of decreasing the effective bandwidth. While we implemented all the engines in Verilog HDL, the synthesis is executed in Vivado 16.4. First, simulations are executed to attest the accuracy of the results. Next, we addressed timing errors until the design meets the timing requirements (166 MHz). Table IV shows the percentage of utilization of the resources in the FPGA per module.

TABLE IV
FPGA RESOURCE UTILIZATION PERCENTAGES PER ENGINE - IMAGE SIZE 320×240

Available Resources	Preprocessing (%)	Cell Desc (%)	Block Desc (%)	Video Desc (%)
Registers (2443K)	15.49	17.33	14.26	21.22
LUTs (1221K)	20.91	24.79	17.96	25.06
LUTRam (344K)	19.87	16.80	16.41	25.20
Block Rams (1.2K)	40.21	42.52	39.36	87.41
DSPs (2.1K)	0.00	20.00	1.48	59.44
Memory Channels (32)	100.00	100.00	100.00	75.00

To analyze the resource utilization per engine, we split the engines into two groups taking as the dividing factor the utilization of memory channels. Group one, the preprocessing, the cell descriptor and block descriptor engine, uses 100% of the memory channels while the second group, the video descriptor engine, uses 75% only. By inspection of the resource utilization table, we notice that engines in the first group have

[2] $5K$ operations due to $P \times \bar{\boldsymbol{g}}_b$, K divisions, K comparisons, $3K$ operations due to squared roots, plus K multiplications.

[3] $7K$ operations are due to additions and $4K$ operations are due to normalizations. The constant c accounts for few additional operations.

high I/O utilization and lower on-chip resource utilization whereas engines in the second group have high on-chip resource utilization and low I/O utilization. In other words, the number of I/O operations constrains the performance of the first group of engines, whereas the number of arithmetic operations constrains the performance of the second group of engines.

B. FPGA Throughput Results

In this section, we describe the performance of the FPGA design. The performance per engine is shown in Table V. In this table, the time to move the data from the host to the FPGA and back is not reported.

TABLE V
VIRTEX-7 FPGA THROUGHPUT PER ENGINE - IMAGE SIZE 320 × 240

Engine	Videos Processed	Throughput (fps)
Preprocessing	8	11,184
Cell Descriptor	4	3,110
Block Descriptor	4	11,186
Video Descriptor	4	3,036
Overall Throughput (fps)	4	1,088

Table V shows that the preprocessing and the block descriptor engines have the highest throughput while the other two engines have the lowest. The high performance of the preprocessing engine is the result of two factors. First, it contains eight kernels with each kernel processing three images in parallel. Second, it benefits of the high I/O performance offered by WX-2000 memory system due to data locality during reads and writes [8]. The use of pipelining increases the throughput further. The block descriptor engine has a high performance as well. Notice this engine is fully constrained by the off-chip bandwidth. In this regard, this engine partially benefits from contiguous memory reads since cell descriptors are represented as 16 contiguous bytes. Writes are always issued in eight-byte chunks.

The cell descriptor engine has the next best performance. The performance of this engine is limited by the sparsity of the off-chip reads. Although the number of I/O operations the engine issues is low, memory requests are issued to non-contiguous memory regions. In addition, because the design uses 100% of the memory channels, further gains in performance by means of increasing the processing pipelines is not feasible. The video descriptor engine has the lowest performance. The performance of this engine is limited by the computational complexity of the matrix multiplication operation, see Table I. Further gains in performance are not feasible as resources have been nearly exhausted, see Table IV.

Overall, when working with images size 320×240, the maximum throughput in steady state is 3,036 fps when four FPGAs are used. In steady state engine one processes four videos. Moreover, not considering reconfiguration time, the maximum throughput achieved by one FPGA is 1,088 fps. This calculation accounts for the time it takes for an image to move across each engine.

C. GPU Throughput Results

In this section, we analyze the performance of the GPU design. The first testbed consists of an Ubuntu workstation equipped with an Intel I7-860 processor, 8GB of RAM, and a $K20$ GPU. The second testbed consists of an Ubuntu workstation equipped with an Intel Xeon E5-520 processor, 24GB of RAM, and a $K40$ GPU. Finally, the third testbed consists of a CentOS workstation equipped with an Intel Xeon E5-2680 processor, 32GB of RAM and a $K80$ GPU. The code is compiled with the CUDA compiler release 7.5 and the Basic Linear Algebra Subroutines. In all the experiments, the error correction capabilities (ECC) are disabled. Table VI shows the throughput per engine for each GPU. The discussion that follows applies to the $K20$ GPU. Similar analysis applies to the $K40$ and $K80$ GPUs as these devices share the same architecture.

TABLE VI
K20, K40 AND K80 THROUGHPUT PER ENGINE WHEN PROCESSING EIGHT VIDEOS IN PARALLEL -IMAGE SIZE 320 × 240

	K20 (fps)	K40 (fps)	K80 (fps)
Preprocessing	3,310	4,044	5,306
Cell Descriptor	13,241	17,143	23,594
Block Descriptor	118,154	128,000	243,810
Video Descriptor	9,458	11,294	16,203
Overall Throughput(fps)	2,033	2,487	3,370

From the table, we notice the $K20$ is very fast at computing the block descriptors, engine three, and very slow at preprocessing the videos, engine one. Two elements provide insight on the performance of engine three. First, the performance of this engine is purely I/O bounded. Second, the $K20$ off-chip memory bandwidth is high, i.e. 208 GB/s. The engine computing the cell descriptors, has the next best performance. Close inspections of the performance of this engine shows the kernel computing the sub-block(cell) descriptors taking 66(34) % of the running time. The performance of this engine is limited by the amount of work the engine must execute and by the uncoalesced nature of the reads during the computation of the sub-block descriptors. Thread divergences present during vector normalizations limit the performance as well.

The performance of engine four, namely the computation of the video descriptors, has the next best performance. Inspecting the performance of this engine reveals that about 60% of the time is spent executing the matrix multiplication. The remaining time is spent in nearly equal parts in the kernels responsible for computing Q_N, $\rho(R,Q)^2$ and $\boldsymbol{x}$. In this engine, elements limiting gains in performance include the complexity of the matrix multiplication (see Table III in Section VII), the use of block barriers, and the use of atomic primitives during the computation of the video descriptor vector $\boldsymbol{x}$.

Notably, the preprocessing engine has the lowest performance. The preprocessing complexity analysis (see Table I in Section VII), explains in part this behavior. When compared with the FPGA complexity, the GPU executes $MN(144 \times 8)$ additional I/O operations. Moreover, while the number of the arithmetic operations per integral video in FPGAs is proportional to $7MN$, this complexity is proportional to $9MN$ in

GPUs. On closer inspection, the GPU running times show that engine one spends 55.0%, 28.9%, 12.2%, and 3.8% computing the row integrals, the columns integrals, the image gradients, and the integral videos respectively. Issues affecting the performance of the row integrals kernel include the presence of *control flow divergences*, the presence of *synchronization primitives*, and the *effective amount of work* that a thread executes per step.

Bialas [49] shows that block thread divergences on Kepler GPUs cost as much as 116 clock cycles. *Letrendre* [50] shows that the extra cost of using block synchronization primitives in the presence of global memory reads ranges from few hundreds up to a thousand cycles. Likewise, the cost of global memory writes in the presence of synchronization primitives is comparable, although it tops at about 700 cycles. The extra cost of using block synchronization primitives in the presence of shared memory reads is near 350 clock cycles. Similarly, when synchronization primitives are used, the cost of shared memory writes is near 220 cycles. Furthermore, the row integral kernel executes $2(N-1)$ additions in $2Log_2(N)-1$ steps when $N/2$ threads are used. If $N=512$, the number of additions per step is $61 \approx 1022/17$. In this case, the amount of work per thread per step is $0.24 \approx 61/256$. In other words, during row integration threads do not execute any useful work 76% of the time.

In addition, we notice that, although recent GPU architectures include novel software and hardware optimizations [51], in our work, those optimizations do not increase the throughout of the row integral kernel notwithstanding the expected gains in performance due to the new architecture. While the single instruction multiple thread (SIMT) execution model supports independent thread scheduling, this model does not increase the performance of the kernel under analysis because synchronization between the collaborating threads during the up-sweep and the down-sweep is still required i.e. in the best scheduling scenario, the integral row kernel still requires $2Log(N)-1$ steps. Further research reveals that the low performance displayed by the row integral kernel is part of a broader set of performance challenges faced by GPUs when processing workloads with irregularities as shown in [52], [53], [54].

We notice that, although the $K20$ GPU can process fifteen videos in parallel, the gains in performance are diminishing as the number of videos increases. The peak performance is achieved when the number of videos processed is ten. Above ten videos, the performance remains constant. Below seven videos, the throughput drops by 30 fps and below. In brief, when working with images size 320×240, the throughput of the $K20$, $K40$, and $K80$ is 2,033 fps, 2,487 fps, and 3,370 fps respectively. The $K80$ speedup is $1.3X(1.6X)$ when compared with the $K40(K20)$. The $K80$ implementation takes advantage of the dual GPU design.

D. Heterogeneous HAR

Based on the throughput results obtained for the $K20$ GPU and the Virtex-7 FPGA, in this Section, we develop a heterogeneous HAR (HHAR) design. In this design, the preprocessing is executed in the FPGA. The data is then moved from the FPGA to the host and from the host to the GPU, and finally, the cell, block, and video descriptors are computed in the GPU. Fig. 11 shows the steps required by our HHAR design.

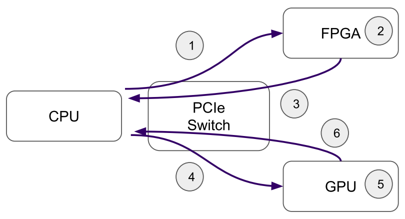


Fig. 11. Heterogeneous HAR desing. (1) The transferring of data from the host to the FPGA (2) The execution in the FPGA (3) The transferring of data from the FPGA to the CPU (4) The transferring of data from the CPU to the FPGA (5) The execution in the GPU (6) The transferring of data from the GPU to the CPU.

Table VII shows the execution times of the steps involved in the algorithm for images size 640×480.

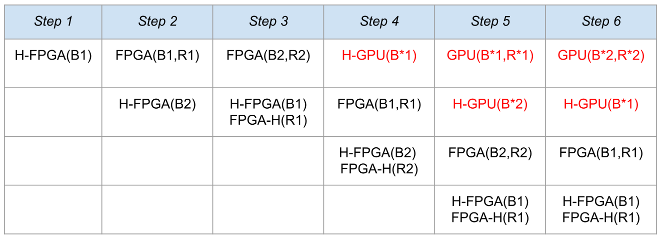
TABLE VII
HETEROGENEOUS HAR DESIGN EXECUTION TIMES PER TASK - IMAGE SIZE 640×480

Step	Resource	Time (ms)
(1) Host to FPGA Data Transfer	PCIe G3 $\times 16$	65.24
(2) Preprocessing	Virtex 7	274.70
(3) FPGA to Host Data Transfer	PCIe G3 $\times 16$	379.16
(4) Host to GPU Data Transfer	PCIe G2 $\times 16$	225.84
(5) Cell, Block and Video Desc.	K20	591.50

Table VII shows the times it takes to process eight videos in parallel. In this design, $776 = 97 \times 8$ gray-scale images are transferred from the host to the FPGA. Next, the preprocessing engine is executed in the FPGA. The resulting 8×144 2-byte integral videos are then transferred from the FPGA to the host and from the host to the GPU. Finally, the GPU executes the cell, block, and video descriptor engines. The time to transfer the video descriptor back to the CPU, step six, is below one millisecond, and as a result, it is not reported.

By inspection of Table VII, we notice that the execution of the block, cell, and video descriptor engines on the GPU takes the longest time followed by the time it takes to transfer data from the FPGA to the host. Moreover, it is possible to overlap the movement of data from the host to the FPGA (and vice versa) with the execution of the kernel in the FPGA; the call $wdm\_dispatch$ in the Convey Development Kit is non-blocking [55]. Likewise, it is possible to overlap the movement of data between the host and the GPU with the execution of a kernel in the GPU given that several practices are observed [42]. Considering these overlaps, the communication time between the host and the FPGA dominates the FPGA execution time. Similarly, the execution time in the GPU dominates the communication time between the host and the GPU.

Based on these observations, we propose a host controlled four stage pipeline, see Fig. 12. In this plot, the notation *Bx,Rx* and *B*x, R*x* identifies the set of double buffers used in the



Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
H-FPGA(B1)	FPGA(B1,R1)	FPGA(B2,R2)	H-GPU(B*1)	GPU(B*1,R*1)	GPU(B*2,R*2)

	H-FPGA(B2)	H-FPGA(B1)
		FPGA-H(R1)
			FPGA(B1,R1)	H-GPU(B*2)	H-GPU(B*1)

			H-FPGA(B2)
			FPGA-H(R2)
				FPGA(B2,R2)	FPGA(B1,R1)

				H-FPGA(B1)
				FPGA-H(R1)
					H-FPGA(B1)
					FPGA-H(R1)

Fig. 12. Heterogeneous HAR Pipeline. The pipeline has four steps: (a) the transferring of data from the host to the FPGA *(H-FPGA)* and from the FPGA to the host *(FPGA-H)*; (b) the execution in the FPGA *(FPGA)*; (c) the transferring of data between the host and the GPU *(H-GPU)*; and (d) the execution in the GPU *(GPU)*.

FGPA and the GPU. In steady state, reached at step six, the maximum latency of the pipeline is 591.50 ms. This is the time it takes the GPU to process eight videos in parallel. Based on these considerations, the HAR design has a cumulative throughout of 1,311 fps; 163 fps per input video.

E. Energy Efficiency Comparison

Next, we compare the energy efficiency of each platform. For the GPUs, the power is measured using the NVidia Management Library [42]. Once the power plot is drawn, corrections have been made to have an accurate power estimation [56]. The FPGA power consumption is measured using the Convey Development Kit [55]. This API allows the user to query the power usage of the FPGA as the application is executed. In all platforms, the computed power accounts for the idle power and the dynamic power consumption. The energy usage and energy efficiency per platform and per engine are shown in Table VIII. Although our heterogeneous HAR design works with any GPU, we report the results with the $K20$ GPU.

TABLE VIII
ENERGY USAGE (JOULES), ENERGY EFFICIENCY (FRAMES/JOULE), AND THROUGHPUT (FPS) PER PLATFORM - IMAGE SIZE 640 × 480

	HHAR	K20	K40	K80
Preprocessing (J)	17.2	74.0	80.3	125.2
Cell Desc (J)	41.0	14.4	13.8	20.5
Block Desc (J)	4.8	1.7	1.4	2.0
Video Desc (J)	98.9	34.7	35.7	35.6
Total Energy (J)	161.9	124.8	131.2	183.3
Efficiency (F/J)	8.0	3.6	3.9	5.4
Throughput (FPS)	1,304	455	517	998

For the preprocessing stage in the HHAR design, we report the energy measured via the Convey Development Kit and for all other stages, we report the energy measured via the NVidia Development Kit. To obtain the throughput and energy efficiency of the preprocessing engine in the FPGA, we have synthesized its design for VGA images (640 × 480) and measured the throughput and power consumption. The resource usage is shown in Table IV. Eight engines processing gray-scale images have a cumulative throughput of 2,796 fps while requiring about 36.8 joules i.e. 13.1 mJ/F.

Moreover, the HHAR energy calculation shown in Table VIII does not takes into account the energy used by the host or the PCIe buses. Our HHAR design requires additional energy to move $707.8MB$ from the FPGA memory to the host memory and from the host memory to the GPU memory. Our research indicates this additional energy is minor compared to the energy used by a kernel running in either the FPGA or the GPU. It is estimated that DDR3 memories dissipate approximately 1.5 W/GBit on average and close to 2.5 W/GBit at peak usage [3], [57]. In the case of memory reads, the reading of 32 bits requires close to 620 pJ [3]. Using these figures, we estimate the energy required for reads and writes $707.8MB$ is below one joule. In addition, our experiments reveal that the transfer of 5.6 GBits from the host to the GPU requires about a dozen Watts, as reported by the sensor in the GPU, although precise measures of the energy required bit the PCIe links is challenging. More importantly, adding few joules to the energy consumption of our HHAR design will not alter the overall results.

From this, we notice that the HHAR design has the highest throughout, in frames per second (fps), and it is the most energy-efficient design, in frames per joules (F/J), followed by the K80. Our HHAR design has a cumulative throughput of 1,311 fps: 163 fps for each incoming video. In addition, it achieves $2.0X(2.2X)$ higher energy efficiency when compared with the K40(K20). The K20 and K40 GPUs have comparable comparable energy efficiency and the $K80$ is more energy efficient by a factor of 1.5 and 1.4 respectively. Notice that if our HHAR design uses the $K40$ or $K80$, instead of the $K20$ GPU, our design will further increase both the energy efficiency and the throughout.

F. Comparison With Other Works

Prior work on HOG has focused mostly on two dimensions (HOG2D) for object recognition. Instead, we use histogram of gradients in three dimensions (HOG3D), which is particularly important for HAR. Working with the temporal dimension adds to the *complexity* of the algorithm in all its stages.

Previous research of HOG2D for object recognition in GPUs includes the work presented in [58], [59], [60]. When processing images size 640×480, as in this work, these designs achieve throughputs ranging from 16 fps up to 38 fps. While the focus of the work in [58] is the identification of vehicles in real time, the work in [59], [60] focuses in the identification of pedestrians using batch approaches. In addition, these researchers focus their attention on achieving high throughput and high energy efficiency using well-establish algorithms.

Research of HOG2D for object detection in FPGAs includes [61], [62], [63], [64], [65], [66]. When processing images size 640 × 480, these designs achieve throughputs ranging from 30 fps up to 526 fps although the work in [64] processes higher resolution images at the expense of lower throughput. As in the case of GPUs, the focus of this work is in achieving high throughput. In addition, lowering the computational complexity of the design without sacrificing the recognition accuracy is paramount.

The acceleration of HOG3D has not received the same attention as that of HOG2D. The work in [29] targets HAR applications in FPGAs although it operates at 600 fps while using images size 320×240. This design has a recognition rate of 93.2% working with a small set of actions. Its recognition drops to 80.8% when a few more actions are added. In comparison, our work achieves a throughput of 1,311 fps on 640×480 images when eight videos streams are processed in parallel. Also, while the work in [67] and [29] target datasets having few classes, our work targets datasets having over 50 classes.

Furthermore, although our work is orthogonal to those focused into improving the accuracy of HAR applications, we state that our HCF design, with multiple scale support, has recognition accuracy comparable to state-of-the-art CNNs. In the case of the HMBD-51[4] [68], the recognition accuracy of CNNs [20] is 59.4% when two-stream CNNs are used. When only the temporal or spatial stream is used, the recognition accuracy drops to 54.6% and 40.5% respectively. When hybrid approaches are used [26], the recognition accuracy reaches 65.9%. Our 16-bits reduced fix-point HHAR design achieves 60.1% recognition accuracy in the UCF50.

Finally, the higher accuracy demonstrated by CNNs on HAR applications [69], [19], [70] comes at the cost of higher power consumption and lower throughput. The results in [4] show that feature extraction using HOG is $311X$ and 13,486X more energy efficient and has $34.7X$ and 1,562X higher throughput than AlexNet [69] and VGG-16 [71] respectively. The work in [5] shows that a five-layer CNN has comparable accuracy to those of HOG designs while consuming $100X$ more energy. In addition, our experiments show that our hybrid design is $44.7X$ more energy efficient and achieves $13.4X$ higher throughput than AlexNet on the Titan X GPU [72].

IX. Conclusions

In this paper, we have investigated the throughput and energy efficiency of HOG3D-based HAR applications acceleration on FPGAs and GPUs for edge computing where high performance and energy economy are at a premium. We have identified four stages in this application and have explored the design constraints of each stage on the target platforms. We have developed a detailed I/O and computational complexity analysis of each of these stages and used this insight to guide our heterogenous implementation. Our results show that a heterogeneous implementation where the first stage, the video pre-processing, is implemented on the FPGA and the other three stages are implemented on the GPU achieves the highest throughput and energy efficiency. Specifically, the heterogeneous HAR algorithm achieves $1.3X$ speedup when compared with the $K80$ GPU, $2.5X$ when compared with the $K40$ GPU, and $2.8X$ when compared with the $K20$ GPU. Similarly, our heterogeneous HAR design is $1.5X$ and $2.0X$ more energy efficient when compared with the $K80$ and $K40$ GPUs. We have shown that HOG3D can be implemented via a reduced fixed-point processing pipeline without compromising the recognition accuracy. Additionally, our design has comparable accuracy to those of HAR design using five-layer CNNs while been more energy efficient.

[4]This benchmark is comparable to the UCF50 benchmark. It has 51 action categories and 7,000 video clips

References

[1] S. Herath, M. Harandi, and F. Porikli, "Going deeper into action recognition: A survey," *Image and Vision Computing*, vol. 60, pp. 4 – 21, 2017.

[2] H. Wang, M. M. Ullah, A. Klaser, I. Laptev, and C. Schmid, "Evaluation of local spatio-temporal features for action recognition," in *British Machine Vision Conference*. London, United Kingdom: BMVA Press, 2009, pp. 124.1–124.11.

[3] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *International Conference on Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[4] A. Suleiman, Y.-H. Chen, J. Emer, and V. Sze, "Towards closing the energy gap between HOG and CNN features for embedded vision," pp. 1–4, 2017.

[5] W. Y. Zou, X. Wang, M. Sun, and Y. Lin, "Generic object detection with dense neural patterns and regionlets," *arXiv preprint arXiv:1404.4316*, 2014.

[6] X. Ma, J. M. Rodriguez-Borbon, W. Najjar, and A. K. Roy-Chowdhury, "Optimizing hardware design for human action recognition," in *26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–11.

[7] A. Klaser, M. Marszalek, and C. Schmid, "A spatio-temporal descriptor based on 3D-gradients," in *19th British Machine Vision Conference*, Leeds, United Kingdom, Sep. 2008, pp. 275:1–10.

[8] "Convey wolverine accelerator," https://www.micron.com/, accessed: 2017-09-04.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[10] D. G. Lowe, "Object recognition from local scale-invariant features," in *The proceedings of the Seventh International Conference on Computer Vision*, vol. 2. IEEE, 1999, pp. 1150–1157.

[11] R. McConnell, "Method of and apparatus for pattern recognition," 1986, US Patent 4,567,610. [Online]. Available: https://www.google.com/patents/US4567610

[12] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld, "Learning realistic human actions from movies," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2008, pp. 1–8.

[13] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on Statistical Learning in Computer Vision*, vol. 1, no. 1-22. Prague, 2004, pp. 1–2.

[14] C. Schuldt, I. Laptev, and B. Caputo, "Recognizing human actions: a local SVM approach," in *Proceedings of the 17th International Conference on Pattern Recognition*, vol. 3. IEEE, 2004, pp. 32–36.

[15] P. Dollár, V. Rabaud, G. Cottrell, and S. Belongie, "Behavior recognition via sparse spatio-temporal features," in *Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance*. IEEE, 2005, pp. 65–72.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] H. Jhuang, T. Serre, L. Wolf, and T. Poggio, "A biologically inspired system for action recognition," in *11th International Conference on Computer Vision*. IEEE, 2007, pp. 1–8.

[18] K. Ho-Joon, J. S. Lee, and Y. Hyun-Seung, "Human action recognition using a modified convolutional neural network," in *International Symposium on Neural Networks*. Springer, 2007, pp. 715–723.

[19] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[20] K. Simonyan and A. Zisserman, "Two-Stream convolutional networks for action recognition in videos," in *Advances in Neural Information Processing Systems*, 2014, pp. 568–576.

[21] L. Wang, Y. Xiong, Z. Wang, and Y. Qiao, "Towards good practices for very deep Two-Stream convnets," *arXiv preprint arXiv:1507.02159*, 2015.

[22] C. Feichtenhofer, A. Pinz, and A. Zisserman, "Convolutional Two-Stream network fusion for video action recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 1933–1941.

[23] B. Zhang, L. Wang, Z. Wang, Y. Qiao, and H. Wang, "Real-time action recognition with enhanced motion vector CNNs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2718–2726.

[24] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2625–2634.

[25] L. Sun, K. Jia, D.-Y. Yeung, and B. E. Shi, "Human action recognition using factorized spatio-temporal convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 4597–4605.

[26] L. Wang, Y. Qiao, and X. Tang, "Action recognition with trajectory-pooled deep-convolutional descriptors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 4305–4314.

[27] M. Yang, S. Ji, W. Xu, J. Wang, F. Lv, K. Yu, Y. Gong, M. Dikmen, D. J. Lin, and T. S. Huang, "Detecting human actions in surveillance videos," in *TREC Video Retrieval Evaluation Workshop*, 2009.

[28] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "DeCAF: A deep convolutional activation feature for generic visual recognition," in *arXiv preprint*, 2013, pp. 647–655.

[29] Z. Hou, H. Zhu, N. Zheng, and T. Shibata, "A single-chip 600-fps real-time action recognition system employing a hardware friendly algorithm," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 762–765.

[30] F. Rhodes, "On the metrics of chaudhuri, murthy and chaudhuri," *Pattern Recognition*, vol. 28, no. 5, pp. 745–752, 1995.

[31] J. Liu, J. Luo, and M. Shah, "Recognizing realistic actions from videos in the wild," in *Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 1996–2003.

[32] J. Liu, Y. Yang, and M. Shah, "Learning semantic visual vocabularies using diffusion distance," in *Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 461–468.

[33] K. K. Reddy and M. Shah, "Recognizing 50 human action categories of web videos," *Machine Vision and Applications*, vol. 24, no. 5, pp. 971–981, 2013.

[34] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A dataset of 101 human actions classes from videos in the wild," *arXiv preprint arXiv:1212.0402*, 2012.

[35] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1–27, 2011.

[36] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[37] "Xilinx DSP slices," https://www.xilinx.com/, accessed: 2017-09-04.

[38] K. Gilles, "The semantics of a simple language for parallel programming," *Information Processing*, vol. 74, pp. 471–475, 1974.

[39] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 4, pp. 433–448, 2007.

[40] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the 13th International Symposium on Field-programmable Gate Arrays*. ACM, 2005, pp. 86–95.

[41] V. B. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, "FPGA based high performance double-precision matrix multiplication," *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 322–338, 2010.

[42] "CUDA toolkit documentation," https://docs.nvidia.com/cuda/, accessed: 2017-09-04.

[43] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A work-efficient step-efficient prefix sum algorithm," in *Workshop on Edge Computing Using New Commodity Architectures*, 2006, pp. 26–27.

[44] B. Bilgic, B. K. Horn, and I. Masaki, "Efficient integral image computation on the GPU," in *Intelligent Vehicles Symposium*. IEEE, 2010, pp. 528–533.

[45] "Dense linear algebra on GPUs," https://developer.nvidia.com/cublas/, accessed: 2017-09-04.

[46] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," https://www.cs.colostate.edu/cs675/MatrixTranspose.pdf, accessed: 2018-04-01.

[47] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *Proceedings International Conference on Image Processing*, 2010, pp. 3757–3760.

[48] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[49] P. Bialas and A. Strzelecki, "Benchmarking the cost of thread divergence in CUDA," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2015, pp. 570–579.

[50] J. T. Letendre, *Understanding and modeling the synchronization cost in the GPU architecture*. Rochester Institute of Technology, 2013.

[51] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[52] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[53] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195.

[54] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on FPGAs," in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2011, pp. 31–34.

[55] "Convey PDK reference manual," http://www.conveysupport.com/, accessed: 2017-09-04.

[56] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 2014, pp. 28–36.

[57] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 37–48.

[58] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita, "GPU implementations of object detection using HOG features and deformable models," in *International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 2013, pp. 106–111.

[59] V. Prisacariu, I. Reid *et al.*, "fastHOG - A real-time GPU implementation of HOG," *Department of Engineering Science*, vol. 2310, no. 9, 2009.

[60] K. Lillywhite, D.-J. Lee, and D. Zhang, "Real-time human detection using histograms of oriented gradients on a GPU," in *Workshop on Applications of Computer Vision*. IEEE, 2009, pp. 1–6.

[61] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura, "Hardware architecture for HOG feature extraction," in *Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. IEEE, 2009, pp. 1330–1333.

[62] K. Negi, K. Dohi, Y. Shibata, and K. Oguri, "Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm," in *International Conference on Field-Programmable Technology*. IEEE, 2011, pp. 1–8.

[63] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "An FPGA implementation of a HOG-based object detection processor," *IPSJ Transactions on System LSI Design Methodology*, vol. 6, pp. 42–51, 2013.

[64] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-based real-time pedestrian detection on high-resolution images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 629–635.

[65] V. Ngo, A. Casadevall, M. Codina, D. Castells-Rufas, and J. Carrabina, "A high-performance hog extractor on FPGA," *arXiv preprint arXiv:1802.02187*, 2018.

[66] X. Ma, W. A. Najjar, and A. K. Roy-Chowdhury, "Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs," *Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 6, pp. 1051–1062, 2015.

[67] A. Al Maashri, M. Debole, M. Cotter, N. Chandramoorthy, Y. Xiao, V. Narayanan, and C. Chakrabarti, "Accelerating neuromorphic vision algorithms for recognition," in *Design Automation Conference (DAC)*. IEEE, 2012, pp. 579–584.

[68] H. Kuehne, H. Jhuang, R. Stiefelhagen, and T. Serre, "HMDB51: A large video database for human motion recognition," in *High Performance Computing in Science and Engineering*. Springer, 2013, pp. 571–582.

[69] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2012, pp. 1097–1105.

[70] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 4489–4497.

[71] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[72] "GPU-based deep learning inference: A performance and power analysis," https://https://www.nvidia.com/, accessed: 2018-04-01.