2015 IEEE 18th International Conference on Intelligent Transportation Systems

libHOG: Energy-Efficient Histogram of Oriented Gradient Computation

Forrest N. Iandola[1,2], Matthew W. Moskewicz[1], and Kurt Keutzer[1,2]
[1]University of California, Berkeley
[2]DeepScale
{forresti, moskewcz, keutzer}@eecs.berkeley.edu

***Abstract*—Histogram of Oriented Gradients (HOG) features are the underlying representation in automotive computer vision applications such as collision avoidance and lane keeping. In these applications, we have observed that HOG feature computation is often a slow and energy-intensive component of the overall pipeline. In this paper, we focus on reducing both the time taken and the energy used for computing Felzenszwalb HOG features. We achieve our results though a combination of reduced precision, SIMD parallelism, algorithmic changes, and outer-loop parallelism. In particular, we address a bottleneck in histogram accumulation by phrasing the problem as a gather instead of the (traditional) scatter. Additionally, we explore the tradeoffs of using L1 instead of L2 norms to compute gradients, which enables smaller operands and more SIMD parallelism. Overall, we are able to compute multiresolution HOG pyramids at 70fps for 640x480 images on a multicore CPU. This is a 3.6x speedup over the best known HOG implementation and a 29x speedup over the popular voc-release5 HOG code. This is also a 3.6x - 22x reduction in energy per frame compared to previous HOG implementations. Our open-source implementation is available for download.**

I. Introduction

Histograms of Oriented Gradients (HOG) are a popular feature representation in computer vision algorithms. HOG is particularly ubiquitous in advanced driver assistance systems (ADAS). For example, [2] used HOG as the underlying feature representation in a lane departure system. In addition, [3] designed a traffic sign detection algorithm on top of HOG features. Tawari et al. used HOG in tracking head and eye movement to monitor driver alertness [4]. Finally, [5] used HOG features to detect pedestrians and vehicles as part of a collision avoidance system.

Real-time computation is crucial in ADAS applications. It is not very useful to detect that the car has drifted out of the lane, if the car has crashed by the time the computer vision system has identified the lane departure. Traffic light detection is of limited value if the vehicle has already violated a red light by the time the vision system has detected the light. ADAS systems typically must run at a speed of at least 30 fps, and HOG is only one of several computations performed per frame. Therefore, it should come as a surprise that the fastest publicly-available HOG implementation (`FFLD` [6]) runs at just 20 frames per second. In this paper, we propose libHOG, which runs at 70 fps on a commodity CPU. This is fast enough for real-time usage in ADAS applications ranging from lane identification to pedestrian and vehicle detection.

Energy efficiency is also important in ADAS. While many DARPA Grand Challenge vehicles used auxiliary generators to power multiple high-end computers, this approach is clearly undesirable for consumer automotive applications. In order to compare energy efficiency across implementations, one could consider simply measuring total system power during operation. However, without normalization due to the differing frame rates of various implementations, total system power can be a misleading metric. Instead, we use the metric of *energy per frame* or J/frame. Our libHOG implementation requires just 2.6 J/frame, which is 3.6x less energy than the previous state-of-the-art.

The rest of the paper is organized as follows. In Section II, we survey other HOG computation methods from the related literature. Next, in Section III we review the general procedure for calculating of HOG features. Then, in Section IV we describe how we accelerate HOG computation. Finally, in Section V, we evaluate the overall speed and energy of libHOG and the accuracy of an object detector using libHOG.

II. Related Work

Histograms of Oriented Gradients (HOG), pioneered by Dalal and Triggs [7], are an extremely widely-used method for feature extraction. A number of HOG variants have been developed over the years, such as Felzenszwalb HOG [1], Circular Fourier HOG [8], Motion Contour HOG [9], and Compressed HOG [10]. Felzenszwalb HOG is used in numerous object recognition methods including Deformable Parts Models [1], Poselets [11], and Exemplar SVMs [12]. The main difference between the Dalal HOG [7] and Felzenszwalb HOG [1] is that Felzenszwalb HOG has a special normalization scheme (see Section IV-C). The canonical reference implementation of Felzenszwalb HOG is in the `voc-release5` [13] Deformable Parts Model codebase in `features.cc`. This implementation is included in numerous open-source computer vision projects, such as the vehicle tracking implementation in [14].

In the literature, we have observed a few attempts to accelerate HOG feature computation. Each of the following implementations produce Felzenszwalb HOG features. First, Dollár's `fhog` [15] exploits SIMD vector parallelism to run faster than the canonical voc-release5 implementation. Next, `FFLD` [6], [16] exploits outer-loop parallelism across image scales, but without SIMD parallelism. Finally, while Dollár and FFLD run on multicore CPUs, `cuHOG` [17] runs on NVIDIA GPUs.

While we focus on Felzenszwalb HOG extraction, there is also some work on accelerating Dalal HOG extrac-

978-1-4673-6596-3/15 $31.00 © 2015 IEEE
DOI 10.1109/ITSC.2015.205

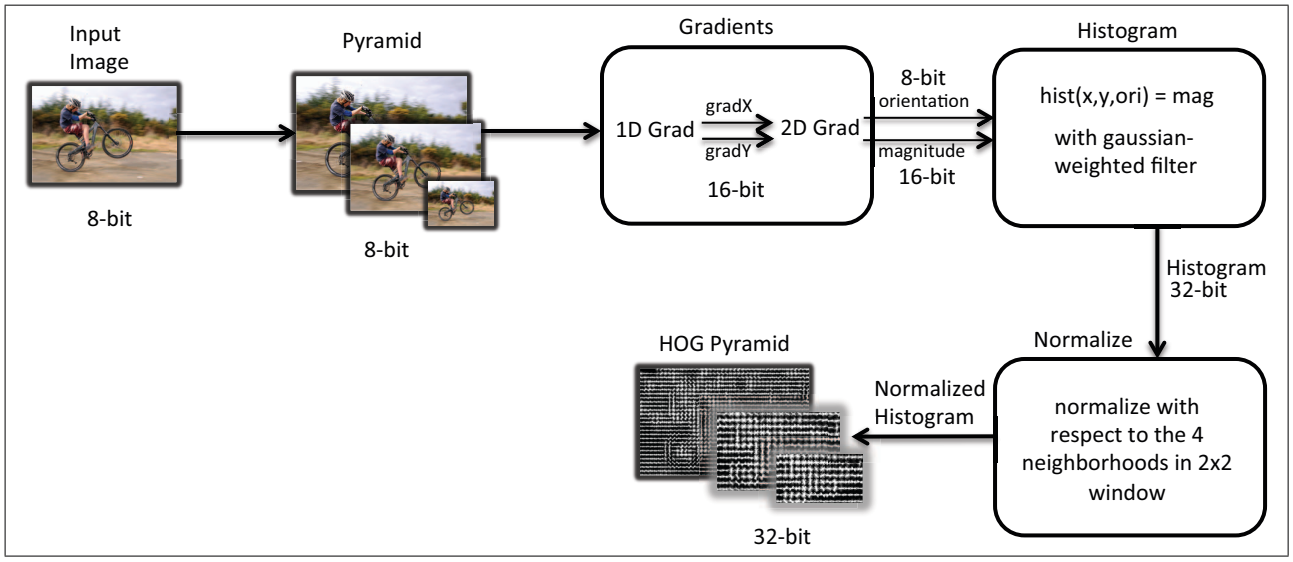

Fig. 1. **Our fast, energy-efficient HOG pipeline.** This produces Felzenszwalb [1] HOG feature maps. We only show 3 HOG pyramid resolutions here, but a typical HOG pyramid in [1] has 40 or more resolutions.

```
calc_hog(image):
  grad_mags, grad_orients = calc_gradients(image)
  grad_histogram = bin_grads(grad_mags, grad_orients)
  return normalize_histogram(grad_histogram)
```

Fig. 2. Per-image HOG feature computation high-level pseudo code

tion. OpenCV [18] provides modestly optimized implementations of Dalal HOG extraction for both CPUs and GPUs. `groundHOG` [19] and `fastHOG` [20] both produce Dalal HOG features using NVIDIA GPUs. There are also a number of HOG implementations that use more exotic hardware such as FPGAs or custom silicon; several of these are surveyed in [21].

For the rest of the paper, we compare our work with best-of-breed approaches: `voc-release5` (widely-used baseline), Dollár (vectorized CPU), `FFLD` (multithreaded CPU), and `cuHOG` (best-of-breed GPU implementation).

III. Introduction to HOG Features

A. Single Image HOG

A single HOG feature image is produced from a single input RGB or grayscale image. Intuitively, HOG features contain information about the spatial distributions of the gradient (i.e. edge) orientations and magnitudes in the input image. There are many variants of HOG features, but they generally share certain key properties from Dalal's original work: quantized orientations, spatial pooling, and local contrast normalization [7]. Pseudo-code for the general computation of HOG features is shown in Figure 2. Although readers are assumed to be familiar with HOG features, we begin with a general, but moderately detailed, operational description of each function in the pseudocode. Although there are no doubt many possible ways to compute HOG features, the description we give here forms a basic outline from which to explain the details of various specific implementations. In particular, we will describe a reference implementation, our best new high-speed implementation, and various other possible design choices and experimental implementations we have tried.

Returning to Figure 2, we will now give a general operational description of how to calculate HOG features for a single grayscale image (noting how the procedure extends to RGB images). Much of the flow is generic with respect to the specific type of HOG feature being computed. However, in this work we consider only Felzenszwalb HOG Features [1] unless otherwise noted. Thus, some of the details of the computation are specific to Felzenszwalb HOG.

First, the $calc\_gradients()$ function computes 1D gradients in the X and Y dimensions by applying the standard centered derivative filter `[-1 0 1]` (for the X gradient) and its transpose (for the Y gradient) at each pixel of the input grayscale image. Then, the magnitude and orientation (or angle) of the (x,y) gradient vector is computed for each pixel. Orientations are quantized to 18 discrete angle values. Note that these 18 orientations consist of 9 pairs of orientations that differ only by the sign of their magnitude (or equivalently by a rotation of 180 degrees). If the input image is multi-channel (i.e. RGB as opposed to grayscale), $calc\_gradients()$ is called for each color channel, and then for each pixel the maximum gradient value across color channels (and its corresponding orientation) are selected and computation then proceeds as in the grayscale input image case.

Next, the $bin\_grads()$ function smoothly spatially bins the per-pixel gradient magnitudes and orientations at a lower resolution than the original image. Typically, the resolution of the gradient histogram is either 1/8 (or sometimes 1/4) that of the input image. Each histogram bin is formed by an approximately-Gaussian weighted pooling over the 16x16 (or 8x8 in the 1/4 scale case) window of gradient pixels centered over the bin.

Finally, the $normalize\_histogram()$ function computes the final per-bin output HOG features using various local normalizations and combinations of the contents of the "raw" or un-normalized gradient histogram just computed by $bin\_grads()$. Note that computation of the final features is independent for each bin. For each bin, we consider the 4 possible 2x2 sub-windows of the 3x3 window of bins centered over that bin. Based on the total per-bin non-normalized gradient energy in each of the 4 sub-windows we compute 4 local "directional" (+X+Y, +X-Y, -X+Y, -X-Y) normalization factors; the average of these 4 directed normalization factors is the bin normalization factor BNF. Then, a total of 31 features are computed as follows: 18 features are computed by taking BNF times the 18 per-bin non-normalized gradient magnitudes; these are termed the contrast-sensitive features. Next, 9 more features are formed by taking BNF times the 9 sums of pairs of magnitudes of the 9 per-bin pairs of 180-degree-rotated orientations; these are termed the contrast-insensitive features. Finally, the last 4 features are created by summing recomputations of all 27 prior features using each of 4 "directional" bin-local normalization constants individually instead of BNF.

B. Existing Implementation Details

In the following sections we will present tables that, for each specific stage of computing HOG features, compare our new algorithm to various existing algorithms. However, here we first highlight the key overall design choices in the other HOG implementations that we have cited and with which we will compare our computational efficiency.

First, we consider the baseline/reference `voc-release5` [13] HOG implementation. For the most part, this implementation is a straight-forward standard C elaboration of the above description. No SIMD or process/thread parallelism is used. A mix of 32-bit $float$ and 64-bit $double$ datatypes are used. Gradient normalization uses the standard L2 norm. In $calc\_gradients()$, orientation quantization is accomplished by taking the dot product of the (x,y) gradient vector with 18 reference orientation vectors and choosing the maximum. In $bin\_grads()$, the gradient pixels are iterated over and added to each histogram bin that they influence. We term this a "scatter" style approach as illustrated in Figure 3.

Dollár's `fhog` [15] implementation improves over `voc-release5` primarily in its use of SIMD parallelism. For determining the quantized gradient orientation, it uses an $arccos()$ lookup-table (or LUT). It also uses a "scatter" style $bin\_grads()$. The `FFLD` [6], [16] implementation uses thread-level parallelism across image scales, but without SIMD parallelism. For determining the quantized gradient orientation, it uses an $arctan()$ lookup-table (or LUT). Like the prior two implementations, it also uses a "scatter" style $bin\_grads()$. `cuHOG` [17], as a GPU based implementation, uses GPU-style SIMT parallelism rather than the SIMD and coarse thread parallelism used by the other (all CPU-based) algorithms discussed here. As it uses the CUDA programming environment, it runs only on NVIDIA GPUs.

C. Multiple Image HOG and Image Resizing

So far, we have described how to compute HOG features for a single image. However, in practice it is often desired to compute HOG features at many scales. This can be accomplished by computing many HOG feature images from many rescaled copies of an input image. Note that the computation HOG features is independent for each scale. In this work, we primarily consider a common case where 10+30 scales of a 640x480 size input image are desired. First, 30 1/8-bin-resolution HOG feature images are computed from 30 progressively downsampled versions of the input image, with 10 equally-logarithmically-spaced scales for every factor of 2 (octave) of downsampling. Then, an additional 10 1/4-bin-resolution HOG feature images are computed from reusing the first 10 largest-scale copies of the input image; the resultant 10 HOG feature images are the same size as would be the first 10 1/8-bin-resolution HOG feature images computed from a 2x upsampled version of the input image.

The HOG implementations in the related work (`voc-release5`, `FFLD`, and Dollár) each have custom, from-scratch implementations of image resizing. There are also a number of off-the-shelf image resizing implementations, found in libraries such as OpenCV [18] and Intel Performance Primitives (IPP) [22]. In Table I, we compare the speed of image pyramid computation using OpenCV, IPP, and the HOG related work. For OpenCV and IPP, we show two variants: a serial version, and a version with OpenMP parallelism across image scales. We found that the image resizing functions in OpenCV and Intel Performance Primitives (IPP) are quite efficient. In libHOG, we use the IPP image resizing function with parallelism across scales. One interesting note is that the majority ($\sim$80%) of the time spent on resizing is on the 10 largest image scales. This is sensible since these largest scales contain $\sim$80% of the total pixels across all scales.

Additionally, while it is possible to exploit thread-level parallelism within a single image, as with `FFLD` we also choose to exploit thread-level parallelism across scales by default.

IV. Our HOG Algorithm

In this section we explain our techniques to accelerate per-image HOG feature computation. As per Figure 2, there are three major steps: gradient computation ($calc\_gradients()$), histogram accumulation ($bin\_grads()$), and normalization ($normalize\_histogram()$). To achieve high speed, we follow two simple high level design principles: First, we wish to minimize memory traffic to avoid being bound by communication costs. Second, we wish to maximize vectorization to avoid being bound by maximum rate at which computations can be issued.

One key technique that helps with both of these issues is to use the most narrow data type possible at each stage of the computation. Sometimes, this involves storing a narrower data in memory and expanding it on the fly for computation. Another important general technique is that of composition. In general, it is best to perform as much computation as is possible for a given data item, or in a given local area of data, before writing it back to memory. As a toy example, when calculating the L2-norm magnitude of (x,y) per-pixel gradients,

TABLE I. **Image Resizing with bilinear interpolation.** COMPARISON WITH RELATED WORK. THIS IS SIMPLY "MAKING 40 COPIES OF A 640X480 INPUT IMAGE AT VARIOUS RESOLUTIONS." EACH EXPERIMENT IS THE AVERAGE OF AT LEAST 100 RUNS.

	Precision	Runtime	Frame Rate
voc-release5	64-bit double	0.16 sec	6.25 fps
Dollár	32-bit float	0.045 sec	22.2 fps
FFLD-serial	8-bit char	0.076 sec	13.2 fps
FFLD-OpenMP	8-bit char	0.016 sec	62.5 fps
OpenCV-serial	8-bit char	0.018 sec	55.5 fps
OpenCV-OpenMP	8-bit char	0.0049 sec	204 fps
Intel IPP-serial	8-bit char	0.0071 sec	141 fps
Intel IPP-OpenMP	8-bit char	0.0023 sec	435 fps

it would be superior to calculate $mag = sqrt(x^2+y^2)$ for each pixel rather than first computing $mag2 = x^2+y^2$ for each pixel and then computing $mag = sqrt(mag2)$ in a second pass.

Note: all reported results are for computing HOG features at 10+30 scales for 640x480 input images on a 6-core Intel i7-3930k processor unless noted otherwise.

A. Gradient Computation

Calculating 1D gradients in X and Y. First, as noted in the resizing discussing, our implementation stores the resized images using the same 8-bit unsigned integer values per-pixel-per-color as the input image. Thus, the input pixels have a range of $[0, 255]$, and the resultant X and Y derivatives have a range from $[-255, 255]$. We choose the simple option of storing the X and Y components of the gradient each as 16-bit signed integers. Thus, the input of the gradient calculation is 1 byte per pixel, and the output is $2 * 2 = 4$ bytes per pixel. To perform the actual calculation, we first load the input pixel data, widen it, and then compute the gradients using 128-bit-wide packed-16-bit-SIMD SSE (hereafter $128w16si$) operations. This allows us to perform 8 concurrent arithmetic operations per CPU clock cycle.[1]

Magnitude. Next, we use the X and Y gradients to compute the per-pixel gradient magnitude. At this stage, our gradients are stored as 16-bit signed integers, albeit with a limited range of $[-255, 255]$. However, when computing the L2 norm, the expression $gradX^2 + gradY^2$ can still (just) overflow a 16-bit integer, and current CPUs do not support vectorized 16-bit floating-point math. Thus, we use 32-bit intermediates for the L2 magnitude calculation to avoid overflow.

Additionally, although it has different semantics, we also chose to experiment with using the L1 norm $|gradX| + |gradY|$ instead of the L2 norm. The output range of this expression is $[0, 510]$, which still easily fits within a 16-bit signed integer.[2] For the L1 norm calculation we again use $128w16si$ operations.

Handling RGB images. As previously mentioned, for RGB images we must calculate the gradient for all three color channels and select the one with the maximum magnitude. Frustratingly, SSE instructions do not natively have the ability to compute the needed argmax operation across 3 channels. Thus, we implement our own vectorized argmax primitive, again using $128w16si$ operations. In summary, we iterate over the 3 per-channel magnitudes and compare them against the largest magnitude seen so far for this pixel. If it is larger than the best seen magnitude, we both replace the best seen magnitude with the current channel magnitude as well as store the corresponding X and Y gradients from the current channel for later use. Standard SIMD comparisons and bit-wise Boolean operations are used to perform these operations in a fully vectorized manner; consult the code for more details.

Orientation. The non-quantized gradient *orientation* for each pixel is defined as $atan2(gradY, gradX)$. However, there does not currently exist a vectorized $atan2$ instruction. Also, $atan2$ is typically a relatively expensive operation. Further, we need only the 18-levels-quantized angle, which allows for various possible optimizations. Currently, we choose to use a modestly sized look-up table: `atan2[gradX][gradY]`. Given that gradX and gradY both have a range of $[-255, 255]$, a LUT with 512*512 entries is sufficient. Further, the elements of the table need only have a range of [0,17] for the 18 quantized orientations, so 1 byte per entry is sufficient. Thus, the total LUT size is only 256 KB. We also experimented with using a directly vectorized version of the voc-release5 orientation computation method, but determined that the LUT-based method was faster to compute.

In Table II, we compare our gradient (orientation and magnitude) computational efficiency with previous HOG implementations. We find that our gradient implementation is 39x faster than voc-release5 and 4.1x faster than the fastest known implementation.

B. Histogram Accumulation

Histogram computation. Recall that the gradient histogram consists of both spatial binning (at 1/8 or 1/4 the input image resolution) and orientation binning. We choose to iterate over the spatial dimension first. At each spatial location, the histogram to be computed by $bin\_grads()$ has 18 orientation bins to fill in, one for each quantization level of the orientations; we term this set of 18 orientation bins a "spatial bin". The contribution of each gradient pixel to each bin is weighted by an approximately-Gaussian function centered over the spatial bin with a width of 8 (or 4 in the 1/4 resolution case) pixels. The spatial bin includes only contributions from the 16x16 (or 8x8 in the 1/4 resolution case) gradient pixels nearest to its center; note that this range is somewhat arbitrarily chosen, but the intent is that pixels further away would not significantly contribute to the bin due to having low weights

[1] We also considered 256-bit AVX instructions. We wrote some HOG-like microbenchmarks in AVX, and we saw no speedup for AVX over SSE. We speculate that 128-bit SSE computation combined with OpenMP parallelism is sufficient to saturate the available memory bandwidth.

[2] In Section V, we will show that training Deformable Parts Model object detectors on HOGs with L1 norm magnitude produces accuracy similar to typical L2 norm magnitude.

TABLE II. **Gradient Computation.** COMPARISON WITH RELATED WORK. IN THE RELATED WORK, ONLY FFLD USES MULTITHREADING. WHERE RELEVANT, WE REPORT RESULTS WITH AND WITHOUT OPENMP MULTITHREADING. (640X480 IMAGES, 10+30 PYRAMID RESOLUTIONS.)

	Vectorization	Precision	Magnitude Calculation	Orientation Binning	Frame Rate
voc-release5	None	32-bit float	L2 norm	iterative arctan LUT	6.25 fps
Dollár	SSE	32-bit float	L2 norm	arccos LUT	30.3 fps
FFLD-serial	None	32-bit float	L2 norm	arctan LUT	15.1 fps
FFLD-OpenMP	None	32-bit float	L2 norm	arctan LUT	58.8 fps
libHOG-L2-OpenMP (ours)	SSE	32-bit int & float	L2 norm	arctan LUT	**143 fps**
ibHOG-L1-serial (ours)	SSE	16-bit int	L1 norm	arctan LUT	102 fps
libHOG-L1-OpenMP (ours)	SSE	16-bit int	L1 norm	arctan LUT	**244 fps**

```
for(mX, mY) in magnitude array:
  hX_ = (mX-sbin/2) / sbin
  hY_ = (mY-sbin/2) / sbin
  ori=orientation(mX,mY)
  for(hX, hY) in (hx_:hx_+1) and (hy_:hy_+1)
    xOff = mX - hX*sbin + (sbin/2)
    yOff = mY - hY*sbin + (sbin/2)
    vx = appox_gauss_weight_lut[xOff]
    vy = appox_gauss_weight_lut[yOff]
    hist(hX,hY,ori)+=magnitude(mX, mY)*vx*vy
```

Fig. 3. **Scatter** histogram code (baseline). This *scatters* data from the magnitude array to the histogram.

```
for(hX, hY) in hist:
  mX_ = hX*sbin - (sbin/2)
  mY_ = hY*sbin - (sbin/2)
  for(xOff, yOff) in (0:sbin*2), (0:sbin*2)
    mX = mX_ + xOff
    mY = mY_ + yOff
    ori=orientation(mX,mY)
    vx = appox_gauss_weight_lut[xOff]
    vy = appox_gauss_weight_lut[yOff]
    hist(hX,hY,ori)+=magnitude(mX, mY)*vx*vy
```

Fig. 4. **Gather** histogram code (our approach, which maintains a smaller working set).

as the Gaussian falls off. The actual weighting function used is decomposable into the product of symmetric X and Y terms, and thus can be computed using only a multiply and two lookups into a 16 (or 8 in the 1/4 resolution case) element LUT. Note that we do not attempt to apply SIMD parallelism for this stage.

Optimization: gather instead of scatter. As previously mentioned, implementations like `voc-release5`, `FFLD`, and Dollár use "scatter"-style histogram construction, as shown in Figure 3. In our experiments, however, we found that instead using a gather-style method yielded faster computation times, particularly in the multi-threaded case. Our theory is that the memory accesses used by scatter-style histogram construction are the limiting factor for the speed of $bin\_grad()$ in the multi-threaded case, regardless of the level (over scales or within a single image) at which multi-threading is used. For each gradient pixel, a scatter-style histogram must write to four spatial bins, yielding an output working set of 2x2x18 on which it is performing gradient accumulation, consisting of sparse random read/writes to the 18 orientation bins of each of the four spatial bins. In contrast, for the gather-style approach, we need only a single spatial bin (18 orientation bins) as our output working set, and we will perform all needed writes to it within a small time window. We illustrate the gather-style approach in Figure 4. While the gather-style approach does also need an 8x8x2 input working set, this set is read-only, has a simple access pattern, and is substantially shared among spatial bins. Thus, overall it seems plausible that the gather-style histogram significantly reduces write overhead while not overly increasing read memory bandwidth or cache usage.

In Table III, we find that our histogram implementation is 8.6x faster than voc-release5 and 1.2x faster than the best known implementation. Note that our L2-norm histogram numerical results precisely match those of voc-release5. When using the L1 norm, our numerical results also agree with a version of voc-release5 similarly modified to use the L1 norm.

C. Neighborhood Normalization

Recall that during $normalize\_histogram()$, 4 local "directional" ($(+X+Y)$, $(+X-Y)$, $(-X+Y)$, $(-X-Y)$) normalization factors are needed, each based on computing the average energy of one of the four 2x2 windows of bins that contain the current bin. However, it can be observed that the total number of unique 2x2 windows of bins is roughly the same as the total number of bins; each 2x2 window and its corresponding normalization constant will be used four times in each of the four different orientations. For example, the 2x2 normalization window and resultant normalization constant for the $(+X-Y)$ direction of bin (x, y) is the same for the $(-X-Y)$ direction of bin $(x+1, y)$. Yet, in previous HOG implementations (voc-release5, FFLD, Dollár), the four directional normalization constants are computed for each neighborhood. We avoid this redundant computation by caching the per-2x2-window normalization constants. This yields roughly a 4x reduction in computation for this portion of $normalize\_histogram()$. In Table IV, we find that our $normalize\_histogram()$ is 66x faster than voc-release5 and 2.9x faster than the fastest previous HOG implementation.

V. EVALUATION

A. Speed and Energy

In Table V, we show the overall speed and energy footprint of libHOG compared to other HOG implementations. In "libHOG-OpenMP," we parallelize each stage individually, with a barrier after each stage – this is essentially the sum of the timings from Sections IV-A to IV-C. However, in "libHOG-OpenMP-pipelined," we put one OpenMP parallel loop over all stages in the HOG pipeline, where each thread is responsible for completing a HOG scale from beginning to end. The "pipelined" version also has the advantage that processors can continue to the next stage when finished, rather than waiting on stragglers.

TABLE III. **Histogram Accumulation**, 640X480 IMAGES, 40 PYRAMID RESOLUTIONS.

	Precision	Loop Ordering	Frame Rate
voc-release5	32-bit float	scatter	13.9 fps
Dollár	32-bit float	scatter	35.7 fps
FFLD-serial	32-bit float	scatter	21.3 fps
FFLD-OpenMP	32-bit float	scatter	100 fps
libHOG-serial (ours)	32-bit float	**gather**	45.7 fps
libHOG-OpenMP (ours)	32-bit float	**gather**	**120 fps**

TABLE IV. **Neighborhood Normalization**, 640X480 IMAGES, 40 PYRAMID RESOLUTIONS.

	Precision	Normalization Map	Frame Rate
voc-release5	32-bit float	redundant computation	35.7 fps
Dollár	32-bit float	redundant computation	25.6 fps
FFLD-serial	32-bit float	redundant computation	34.5 fps
FFLD-OpenMP	32-bit float	redundant computation	83.3 fps
libHOG-serial (ours)	32-bit float	**amortized computation**	137 fps
libHOG-OpenMP (ours)	32-bit float	**amortized computation**	**238 fps**

When using L2 gradient magnitude, Table V shows that libHOG is 24x faster than voc-release5, and 3.0x faster than the fastest known implementation. When we use L1 gradient magnitude, we find that libHOG is 29x faster than voc-release5 and 3.6x faster than the fastest known implementation. This is a 3.0x - 22x reduction in energy per frame compared to previous HOG implementations.

B. Accuracy

So far, we have focused on how to make libHOG as computationally efficient as possible. Now, we verify that libHOG works well in an end-to-end computer vision application.

The PASCAL Visual Object Classes (VOC) challenge ran from 2005 to 2012 [23]. A subset of the challenge tasks focused on detecting the occurrence and bounding boxes of 20 types of objects (car, person, dog, ...) in 5000 photographs. The datasets from this challenge are commonly used for the evaluation of object detection methods. Generally, accuracy on the PASCAL datasets are reported in terms of the single-number mean average precision (mAP) across the 20 object categories. We have observed that ADAS work (such as on-road multivehicle tracking [14]) often looks at PASCAL results for inspiration on object detection methods. With this in mind, we evaluate the accuracy of libHOG with the popular Deformable Parts Model (DPM) [1] detector on the PASCAL 2007 dataset. We parallelized the DPM Cascade [24] to run at 20fps on a multicore CPU, including the overhead of computing HOG pyramids in libHOG.

In Table VI, we find that libHOG-L2+DPM produces the same object detection accuracy as voc-release5+DPM. This is expected given that libHOG-L2 and voc-release5 should produce numerically identical HOG features. Recall from Section IV-A that we can achieve an additional speedup by using an L1 instead of L2 norm to compute the gradient magnitude. In Table VI, we find that using the L1 norm degrades accuracy by approximately 2 percentage points. In our libHOG code release, we provide both L1 and L2 HOG implementations, so the reader can select the appropriate accuracy/efficiency tradeoff for their application.

VI. CONCLUSION

HOG feature extraction is a core building block in numerous advanced driver assistance systems (ADAS). Real-time and energy-efficient computation are crucial to real-world deployability of ADAS applications. With this in mind, we have presented libHOG, an open-source HOG implementation that is 3.6x - 29x faster (and 3.6x - 22x more energy-efficient) than previous HOG implementations. We compute HOG pyramids at 71 fps, and this enables ADAS applications like object detection and lane detection to compute HOG pyramids in real-time.

ACKNOWLEDGMENTS

The authors would like to thank Piotr Dollár and Dennis Park for helpful discussions on HOG neighborhood normalization. Thanks to David Sheffield for his advice on HOG gradient computation. Research partially funded by DARPA Award Number HR0011-12-2-0016, plus ASPIRE industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. The first author is funded by the US DOD NDSEG Fellowship.

REFERENCES

[1] P. F. Felzenszwalb, R. B. Girshick, D. A. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," in *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, 2010.

[2] H. Y. Yalic, A. S. Keceli, and A. Kaya, "On-board driver assistance system for lane departure warning and vehicle detection," *International Journal of Electrical Energy*, 2013.

[3] S. Salti, A. Petrelli, F. Tombari, N. Fioraio, and L. Di Stefano, "A traffic sign detection pipeline based on interest region extraction," in *International Joint Conference on Neural Networks (IJCNN)*, 2013.

[4] A. Tawari, K. Chen, and M. Trivedi, "Where is the driver looking: Analysis of head, eye and iris for robust gaze zone estimation," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2014.

[5] C. Keller, T. Dang, H. Fritz, A. Joos, C. Rabe, and D. Gavrila, "Active pedestrian safety by automatic braking and evasive steering," *IEEE Transactions on Intelligent Transportation Systems*, 2011.

[6] C. Dubout and F. Fleuret, "FFLD," `www.idiap.ch/ cdubout/code/ffld.tar.gz`.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition (CVPR)*, 2005.

[8] H. Skibbe and M. Reisert, "Circular fourier-hog features for rotation invariant object detection in biomedical images," in *IEEE International Symposium on Biomedical Imaging (ISBI)*, 2012.

TABLE V. **End-to-end HOG Efficiency**, 640x480 IMAGES, 40 PYRAMID RESOLUTIONS. CUHOG RESULTS ARE CLAIMED IN [17]. ALL OTHER RESULTS WERE PRODUCED BY THE AUTHORS. `libHOG-L2-OpenMP-pipelined` PRODUCES NUMERICALLY IDENTICAL RESULTS TO `voc-release5`.

	Hardware	Frame Rate	Watts (idle: 86.8W)	Energy
voc-release5	Intel i7-3930k 6-core CPU	2.44 fps	140W	57.4 J/frame
Dollár	Intel i7-3930k 6-core CPU	5.88 fps	155W	26.4 J/frame
FFLD-serial	Intel i7-3930k 6-core CPU	4.59 fps	137W	29.9 J/frame
FFLD-OpenMP	Intel i7-3930k 6-core CPU	19.6 fps	185W	9.44 J/frame
cuHOG	NVIDIA GTX560 GPU	20 fps	not reported	not reported
libHOG-L1-serial (ours)	Intel i7-3930k 6-core CPU	12.3 fps	140W	11.3 J/frame
libHOG-L1-OpenMP (ours)	Intel i7-3930k 6-core CPU	52.6 fps	185W	3.52 J/frame
libHOG-L1-OpenMP-pipelined (ours)	Intel i7-3930k 6-core CPU	**71.4 fps**	185W	**2.59 J/frame**
libHOG-L2-OpenMP-pipelined (ours)	Intel i7-3930k 6-core CPU	**58.8 fps**	185W	**3.15 J/frame**

TABLE VI. **Accuracy** FOR PASCAL 2007 OBJECT DETECTION USING HOG IMPLEMENTATIONS WITH DEFORMABLE PARTS MODELS [1].

HOG Implementation	Gradient Magnitude Calculation	Object Detector	Mean Avg Precision
voc-release5	L2 norm	Deformable Parts Model [1]	33.1%
libHOG-L2-OpenMP-pipelined (ours)	L2 norm	Deformable Parts Model [1]	33.1%
libHOG-L1-OpenMP-pipelined (ours)	L1 norm	Deformable Parts Model [1]	31.2%

[9] S. Kohler, M. Goldhammer, S. Bauer, K. Doll, U. Brunsmann, and K. Dietmayer, "Early detection of the pedestrian's intention to cross the street," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2012.

[10] V. Chandrasekhar, G. Takacs, D. Chen, S. Tsai, Y. Reznik, R. Grzeszczuk, and B. Girod, "Compressed histogram of gradients: A low-bitrate descriptor," *International Journal of Computer Vision (IJCV)*, 2012.

[11] L. Bourdev, S. Maji, and J. Malik, "Describing People: Poselet-Based Approach to Attribute Classification," in *International Conference on Computer Vision (ICCV)*, 2011.

[12] T. Malisiewicz, A. Gupta, and A. Efros, "Ensemble of exemplar-svms for object detection and beyond," in *International Conference on Computer Vision (ICCV)*, 2011.

[13] P. F. Felzenszwalb, R. B. Girshick, D. A. McAllester, and D. Ramanan, "voc-release5," cs.berkeley.edu/ rbg/latent/.

[14] H. Niknejad, A. Takeuchi, S. Mita, and D. McAllester, "On-road multivehicle tracking using deformable object model and particle filter with improved likelihood estimation," *IEEE Transactions on Intelligent Transportation Systems*, 2012.

[15] P. Dollár, "Piotr's Computer Vision Matlab Toolbox (PMT)," vision.ucsd.edu/ pdollar/toolbox/doc/channels/fhog.html.

[16] C. Dubout and F. Fleuret, "Exact acceleration of linear object detectors," in *European Conference on Computer Vision (ECCV)*, 2012.

[17] M. Pedersoli, J. Gonzalez, X.Hu, and X. Roca, "Towards a real-time pedestrian detection based only on vision," *Journal of Intelligent Transportation Systems*, under review.

[18] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 2008.

[19] P. Sudowe and B. Leibe, "Efficient use of geometric constraints for sliding-window object detection in video," in *International Conference on Computer Vision Systems (ICVS)*, 2011.

[20] V. Prisacariu and I. Reid, "fastHOG - a real-time gpu implementation of hog," Department of Engineering Science, Oxford University, Tech. Rep. 2310/09, 2009.

[21] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural study of HOG feature extraction processor for real-time object detection," in *IEEE Workshop on Signal Processing Systems (SiPS)*, 2012.

[22] E. Stewart, *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press, 2004.

[23] M. Everingham, L. J. V. Gool, C. K. I. Williams, J. M. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision (IJCV)*, 2010.

[24] P. F. Felzenszwalb, R. B. Girshick, and D. A. McAllester, "Cascade object detection with deformable part models," *CVPR*, 2010.

[25] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CVPR*, 2014.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.

[27] F. N. Iandola, M. W. Moskewicz, S. Karayev, R. B. Girshick, T. Darrell, and K. Keutzer, "DenseNet: Implementing efficient convnet descriptor pyramids," *arXiv:1404.1869*, 2014.

[28] R. B. Girshick, F. N. Iandola, T. Darrell, and J. Malik, "Deformable part models are convolutional neural networks," in *CVPR*, 2015.

[29] W. Ouyang, X. Wang, X. Zeng, S. Qiu, P. Luo, Y. Tian, H. Li, S. Yang, Z. Wang, C. C. Loy, and X. Tang, "DeepID-Net: Deformable deep convolutional neural networks for object detection," *arXiv:1412.5661*, 2014.

[30] P.-A. Savalle, S. Tsogkas, G. Papandreou, and I. Kokkinos, "Deformable part models with CNN features," in *ECCV Workshops*, 2014.

APPENDIX I: OTHER THINGS WE TRIED

We tried composing the three kernels in Section IV, such that each pixel goes from beginning to end without being copied back to memory. This didn't have much impact in terms of speed or energy.

APPENDIX II: RELATIONSHIP BETWEEN HOG AND CONVOLUTIONAL NEURAL NETWORKS

Over the last two years, convolutional neural networks (CNNs) have brought new levels of accuracy to object detection. For common ADAS problems like vehicle detection and pedestrian detection, the CNN accuracy gains have been moderate. However, CNNs offer huge accuracy improvements in recognizing textured objects like plants and specific types of dogs and cats.

Speed is the major downside of CNN-based object detection. For example, the R-CNN [25] object detector operates at roughly 1/10 fps (2000 J/frame) on a GPU, with most of the time spent extracting CNN features.[3] With a few tricks to amortize CNN feature computation, it is possible to accelerate CNN-based object detection to 1 fps (200 J/frame) on GPUs, as discovered independently by [27], [28], [29], and [30]. Even with these improvements, CNN-based object detection is still too slow for many ADAS applications.

[3]Note that we are discussing *localized object detection*. The task of full-image classification without localization can run orders of magnitude faster (e.g. [26]).