Scalable Learning for Object Detection with GPU Hardware

Adam Coates, Paul Baumstarck, Quoc Le, and Andrew Y. Ng

***Abstract*— We consider the problem of robotic object detection of such objects as mugs, cups, and staplers in indoor environments. While object detection has made significant progress in recent years, many current approaches involve extremely complex algorithms, and are prohibitively slow when applied to large scale robotic settings. In this paper, we describe an object detection system that is designed to scale gracefully to large data sets and leverages upward trends in computational power (as exemplified by Graphics Processing Unit (GPU) technology) and memory. We show that our GPU-based detector is up to 90 times faster than a well-optimized software version and can be easily trained on millions of examples. Using inexpensive off-the-shelf hardware, it can recognize multiple object types reliably in just a few seconds per frame.**

I. INTRODUCTION

Achieving high accuracy in object detection tasks for a moderate number of objects is a major challenge in robotic perception. Our interest in object detection stems from our experience with practical applications on the Stanford AI Robot (STAIR), such as taking inventory of a few types of objects in an office environment [1]. While object detection using cameras and other sensors is well studied, it remains notoriously difficult to perform reliably in practice and often involves carefully crafted but fragile models applied to features that are computationally expensive. There are many trade-offs to be made between computational speed and accuracy–yet we want both for a truly deployable system. In this paper, we describe an approach to object detection that achieves speed through the use of highly parallel, scalable algorithms, and achieves accuracy by leveraging large data sets.

Ideally, Moore's Law [2] would mitigate the computational expenses of robot perception on a yearly basis. Unfortunately, single-CPU clock speeds appear to be stagnating. Thus, we cannot rely on clock speed alone to achieve high speed in object detection tasks. Moore's Law continues to hold, but with increased computational power coming in the form of highly parallel architectures. These include multi-core CPUs and, most spectacularly, inexpensive Graphics Processing Units (GPUs) with literally hundreds of cores and enormous amounts of memory on a single card. In addition to growth in computational power we also have high growth of network bandwidth and non-volatile storage capacity that make it feasible to store and transfer extraordinary amounts of data. In order to develop a system that runs faster as newer hardware becomes available, we must choose algorithms that leverage this exponential growth in resources. We believe that the most successful methods in object detection (or computer vision in general, perhaps) may well turn out to be those that scale most easily with these trends, as they will be able to tackle larger and larger problems where other approaches will be impractical.[1]

We not only desire speed but also accuracy from our detector. A key hurdle to developing an accurate detector is the following: training examples are few while opportunities for mistakes in the real world are many. Cluttered background imagery, for example, provides endless varieties of shapes and shades that can easily be mistaken for a target object if that particular background pattern has not been seen before. Our previous system has often suffered from high false positive rates due to this phenomenon. One solution, of course, is to train on very large numbers of negative examples to reduce the probability of seeing a background pattern that has never been seen before. For this approach to work in practice we must build a system that naturally scales to large data sets. The choice of algorithms in this paper is motivated chiefly by their ability to learn from extremely large training sets.

There has also been significant interest in using large data sets in other domains. In image retrieval [4], [5], [6], many systems perform surprisingly well using relatively unsophisticated algorithms. Similar observations have been made for natural language applications [7]. This suggests that we can, in fact, use relatively simple classification algorithms in our detector. Hence, learning from large amounts of data serves the dual purposes of improving classification accuracy while also allowing us to use simpler algorithms.

In summary, our approach to object detection is motivated by the following: (i) Algorithms that scale well with the exponential growth in (parallel) processing power will be able tackle complex problems more effectively (and will improve with time), and (ii) learning from large training sets offers us the opportunity to use simpler algorithms that are more easily implemented, scale better, and simultaneously achieve higher accuracy.

In Section II, we begin by describing the observations and prior work that motivate our design choices. In Section III we then present our object detection approach and describe how our classifiers are trained. Section IV describes our high-speed implementation of the feature computations and

Adam Coates, Quoc Le and Andrew Y. Ng are with the Computer Science Department at Stanford University. {acoates,quocle,ang}@cs.stanford.edu

Paul Baumstarck is with the Electrical Engineering Department at Stanford University. pbaumstarck@stanford.edu

[1]The graphics community, for instance, realized some time ago that "brute-force image-space" methods like the Z-buffer were more scalable and effective than the asymptotically efficient hierarchical methods that predated them [3].

detection algorithm for test time evaluation using graphics hardware. In Section V we conclude with experimental results that demonstrate the accuracy and speed of our system on realistic scenes.

II. DESIGN MOTIVATION

The object detection system we present is designed, foremost, for scalability with computing resources and training set sizes. Specifically, we will leverage the power of GPUs and many-core CPUs to accelerate the computational elements of our system (especially to reduce testing time), and we will utilize large numbers of training examples to achieve good test performance. Our focus on these features is motivated by several prior successes in the computer vision literature.

One well-known result is the work of Viola and Jones [8]. Their use of fast Haar-like features, coupled with a "cascading" set of classifiers for culling out negative examples, demonstrated state-of-the-art performance in face detection at full-motion frame rates. The Haar features chosen in their work are extremely fast and, by virtue of their speed, can be computed in great variety. Surprisingly, however, the Haar features have not seen as much success in detecting arbitrary objects. While fast, the Haar features have limited expressiveness and, it seems, are particularly suited to detecting certain facial features, while failing when applied to more general-purpose detection. In our work, we will use features that, in a sense, are natural generalizations of Haar features. We use features inspired by the work of Torralba et al. [9], and described in [1]. These features are based on dictionaries of image "patches" extracted from positive examples of the target object. They are more expressive than Haar features and at the same time more specialized to the target object class. Moreover, we will see that they are particularly suited to implementation on GPUs.

Harnessing the growth in the computational power of graphics chips has already been explored by computer vision researchers, particularly for computing image features. For instance, the venerable Canny Edge Detector has been implemented on GPUs using the nVidia CUDA SDK [10], as well as Lowe's SIFT descriptor [11].[2] In our work we implement the patch-based features in [9] using GPU hardware, achieving a substantial performance gain. These features are ideally suited to GPU implementation and, thus, turn out to be extremely fast. Indeed, we believe that scalability and speed will allow these features to perform as well as more complex ones, since we can compute them in greater numbers and variety.

The second key feature of our object detection strategy is the reliance on large data sets. Though object detection in full generality remains unsolved, the benefit of large training sets has already been demonstrated. For instance, the contestants in the PASCAL Visual Object Classes Challenge [12] are asked, in one competition, to locate very generic object classes such as "chairs" or "cars" in a variety of scenes using a fixed training set. The difficulty of this task is considerable, and contestants typically achieve average precision of less than 60%. With small training sets, algorithms for these tasks must rely heavily on prior knowledge provided in the form of hand-coded features and carefully tuned parameters. This also suggests that the results will be brittle since the classifiers must rely on assumptions formed from very little data.

Rather than focusing on improved features and algorithms, we instead focus on learning from large datasets, and use off-the-shelf features and algorithms. Some recent work also lends credence to the hypothesis that learning from large amounts of data may allow better generalization and higher accuracy than algorithmic ingenuity alone. The effectiveness of large training sets for image retrieval, for instance, has been observed by several researchers. Though this domain is concerned with somewhat different desiderata than object detection, results using large data sets are promising. In the work of Nister and Stewenius [6], they observe that the use of large data sets allows them to ignore geometric information that had previously been necessary to achieve good performance. Similar interest in large training sets has been shown in object recognition research for training sets with hundreds of thousands of examples [13]. In our work, we will develop a system that scales to tens of millions of examples.

III. DETECTOR LEARNING

We now describe our object detection approach and its implementation for our robot. Specifically, we will detail: (i) briefly, the form of the images input to our system, (ii) the implementation of our patch-based features, and (iii) our boosting-based classifier used for detection.

A. Problem Setup

Consistent with our motivation to learn from large quantities of data, we also want to learn from rich data whenever it is available. Our robot platform is equipped with a typical 640x480 resolution camera that acquires 8-bit gray-scale intensity images. In addition, however, it also uses the "active stereo" system described in [1] to acquire depth images of the scene. Prior work in object detection has demonstrated that depth data, in addition to 2D imagery, improves recognition performance [14], [1]. We also compute the (smoothed) gradient of the gray-scale image intensity and store its magnitude as a new 640x480 image channel. In this image, edges appear as bright pixels and regions of uniform color appear dark. Figure 1 shows a typical 3-channel input to our detection system.

Given an input image, our object detection system builds on the standard "sliding window" approach [15], [16], [17]. We will construct a binary classifier that, for each subwindow of an image, determines whether the target object is contained (tightly) within the window. Given such a classifier, we then evaluate it independently on a series of windows of varying sizes spaced at uniform intervals over

[2]The CUDA SDK allows programmers to run generic C/C++-style code in parallel directly on the GPU cores. We will use the same SDK in our own implementation in Section IV.

(a) (b) (c)

Fig. 1. A typical set of channels from an image captured by our robot's sensors: (a) gray-scale intensity, (b) intensity gradient, (c) depth.

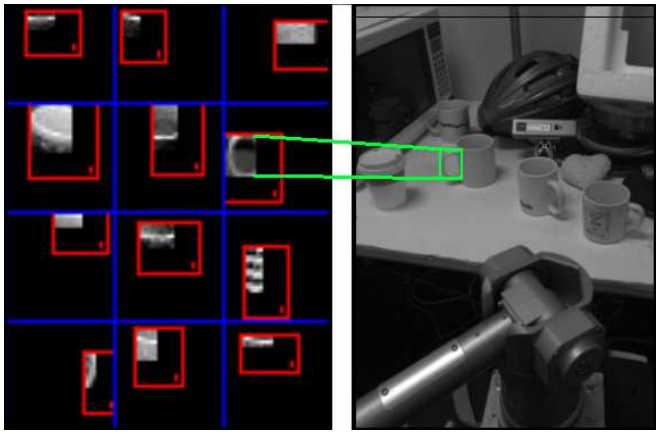

Fig. 2. Examples of patches extracted from a labeled coffee mug. The red rectangles with each patch represent the approximate spatial location of the patch relative to the object center.

the image to detect objects at all locations and scales. While this approach is somewhat brute-force, it is also simple and easy to parallelize on the GPU (as we will do in Section IV), and is also well-suited to our choices of features.

B. Features

Our classification algorithm will operate on a set of features computed from each 3-channel image. We use the patch-based features of [9]. First, a dictionary is constructed from small image fragments. An image fragment g is randomly extracted from image channels in hand-labeled instances of the target object. Each patch is annotated with a rectangle $\mathcal{R}$ specifying its approximate location relative to the object center, and the index c of the image channel from which it was extracted. Specifically, a patch is defined as a triple $\langle g, \mathcal{R}, c \rangle$. Figure 2 shows some examples of patches extracted from examples of coffee mugs along with their associated rectangles.

Given an input image, a patch feature value is computed by first computing the (normalized) cross-correlation of the dictionary patch with the corresponding image channel, and then taking the maximum response over the patch rectangle. More formally, the patch response for a patch $\langle g, \mathcal{R}, c \rangle$ is

$$\max_{i,j \in \mathcal{R}} (\mathcal{I}^c \otimes g)_{i,j} \tag{1}$$

where $\otimes$ denotes normalized cross-correlation, and $\mathcal{I}^c$ is channel c of the input image $\mathcal{I}$.

We have chosen these features because they are general-purpose yet sufficiently specialized that we expect them to work well with a wide range of objects. Since the feature definitions are acquired from data, we can create new features from new data whenever the necessity arises. Thus, as computational resources become greater and training set sizes increase, we can also expand our patch dictionary to more accurately capture the object class structure.

Importantly, these features can also be computed efficiently on GPUs and mesh naturally with the sliding-window approach to object detection. Once we've computed the normalized cross-correlation response for the entire test image, the $\max$ in Equation (1) can be computed for each sub-window independently in parallel. Both the normalized cross-correlation (convolution) and the maximization can be implemented efficiently on GPU hardware.

C. Classifier

Our classification algorithm is also motivated by the same considerations as our features: we use an algorithm that scales well (both in training and testing) when presented with increasing quantities of data, and can cleanly take advantage of the processing power of GPUs at test time, when speed is the most important.

In our work, we use Gentle-Boost [18] with decision trees. Our choice is motivated by a number of observations, not the least of which is that previous work has demonstrated the effectiveness of this particular combination of algorithms in practice [8], [1], [9]. The Gentle-Boost algorithm works by training a set of "weak" classifiers (or "weak-learners") so that the sum of the weak-classifier outputs is a better predictor than the weak-classifiers themselves. Boosting algorithms, including Gentle-Boost, use this capability to generate ensembles of classifiers that are capable of representing extremely complex decision functions. By increasing the number of weak-learners trained by the Gentle-Boost algorithm, we can increase the complexity of the decision function. Thus, when learning from a large data set with complex structure the complexity of our classifier can easily be expanded to meet the challenge.

Our system uses decision trees as weak-learners. This is motivated partly by their compact structure, but also by their (very) sparse dependence on the elements of the feature vector, which has the benefit of reducing the number of features used by the final classifier that is run at test time. It is also possible to train the decision trees in a distributed fashion, which will allow us to train rapidly on very large training sets.

D. Classifier Training

To train our boosted decision trees, we chose to pursue a distributed, parallel approach. There are two main benefits of distributed training: (i) the use of multi-core systems to process the training data repeatedly in parallel to reduce training time and (ii) the ability to leverage the abundance of RAM on multiple machines to hold massive training sets in main memory (thus avoiding expensive disk and network transfers). The Gentle-Boost algorithm is inexpensive by itself and can easily be run on a standard desktop PC for enormous data sets.[3] Thus, the only step that must be distributed is the training of the weak-learners (decision trees, in our case), which we now describe.

Our decision trees are trained similarly to the well-known CART algorithm [19] to minimize the squared error in label predictions (using Gini coefficients as the split criterion). To make our training algorithm distributed, however, we cannot simply compute the Gini coefficient from the entire data set on a single machine. Instead, we use an approximation that has seen success in the data-mining community: we accumulate the feature values for each training example into a histogram, which serves as a sufficient statistic for that feature [20]. Each histogram has 256 buckets (we use fixed bounds, since our features are all normalized to the range [-1,+1]) and thus they are easy to store or transmit, and we only need two histograms per feature (for each machine participating in the training).

In more detail, during training, each worker machine is assigned a subset of the training data and loads those training examples into local memory. Each training example $x^{(i)}$ is associated with a weight $w^{(i)}$ provided by the Gentle-Boost algorithm. The worker then accumulates a weighted histogram for each feature. Thus, bucket B of the histogram for feature j on worker k is computed as:

$$H_j^k[B] = \sum_{i:x_j^{(i)} \in B} w^{(i)}.$$

This is done separately for the positively and negatively labeled training examples, resulting in two histograms for each feature. The resulting histograms for each worker are sent to a master machine where they are summed together to yield two histograms for each feature:

$$H_j[B] = \sum_k H_j^k[B].$$

[3]The only significant computation that is necessary is the update of the weights after each round. This cost scales only linearly with the data set size and is, by a constant factor, quite small.

(Again, one histogram for positive and another for negative examples. Note that these histograms are the same as would be computed on a single machine operating on the entire training set.) The resulting histograms allow us to compute the Gini coefficients for the distribution of feature values and, thus, choose the best split for the node. It is easy to generalize this procedure to training full trees. The only approximation in this procedure is the quantization of the histograms—the algorithm is otherwise identical to running boosting on a single machine and will compute an (approximately) optimal solution.

We have available to us a 32 processor-core cluster (8 machines), with 2GB of RAM available per core (8GB per machine). Each training example is stored on our distributed file system with the features values quantized to 8-bit integers.[4] Thus, using the procedure above, we can accommodate over 60GB of training data or, assuming a 1000-dimensional feature vector, more than 60 million training examples. In addition, we have achieved over 25-fold speedups over single-machine training by running on 32 cores simultaneously.[5]

IV. REAL-TIME TESTING

To achieve real-time detection rates we implemented the major test-time components of our system using nVidia's CUDA GPU development library. The computational advantages of GPUs over CPUs are formidable, and seem unlikely to dissipate. Unlike CPUs, which devote a large number of transistors to large, deep cache systems, GPU architectures have large numbers of computational units with small, shallow caches. GPU pipelines are also optimized for parallel operations using vector processing units, which are naturally suited to computing a single function in parallel over several pieces of data at once. Thus, while GPUs and CPUs may have similar numbers of transistors (with that number growing according to Moore's Law), GPUs allocate these transistors in a way that trades generality for increased computational throughput [21].

In order to achieve high performance using GPUs, we must use algorithms that appeal to the strengths of these architectures while avoiding their shortcomings. We must maximize parallelism (since GPUs are optimized for such operations), data locality (since the GPU caches are small), and, it turns out, minimize memory transfers to and from the device. In general, GPUs are best suited to "data parallel" operations where a single piece of code is executed many times in parallel across multiple pieces of data. The main components of our object detection system share this characteristic: Convolutions can be viewed as many pixel-wise multiplications executed on overlapping sub-windows, and our classifier can be run in parallel over independent sub-windows of the test image. We now discuss the implementation of our feature computations and classifiers on the GPU in more detail.

[4]The loss of precision in storing our features this way is irrelevant since the histograms used during training have only 256 bins.

[5]At the time our experiments were performed, we did not have a GPU-equipped computing cluster, and thus CPU-based training on the cluster was faster than using a single machine with a GPU.

The main computational cost of our detection algorithm is performing the correlations of the test image with the 2D image fragments from the patch dictionary, as described in Section III-B. The patches come in sizes of 4-by-4 pixels up to 16-by-16 pixels. Each patch is correlated with the entire scene, yielding a response image containing (normalized) cross-correlation values for every possible position of the patch in the scene. This "embarrassingly parallel" operation is naturally implemented through the CUDA library. Both the patch and a small portion of the test image can be loaded into the cache of the GPU processors. Once in the cache, the correlation operation can be performed very quickly (using brute-force multiplication and summation) due to the highly data-parallel nature of the computation. After computing the normalized cross-correlation values, it is similarly easy to compute the necessary maximization of Equation 1 in parallel for every window on which the classifier will be evaluated.

Implementing the feature computations alone leads to a noticeable speed up. Unfortunately, this also reveals a critical bottleneck: memory bandwidth. Our source images are 640 by 480 pixels and hence occupy 1.2MB of memory when converted to single-precision floating-point. Meanwhile, a typical CUDA-capable GPU has host-to-device memory bandwidth of roughly a few megabytes per millisecond. Thus the cost of simply copying a full response image back to the host after performing a correlation on the device can severely discount the raw GPU speedup, since we must copy a full response image (itself over 1MB in size) for each feature. In order to achieve higher speeds, naive implementation of expensive computations on the GPU is not enough: one must also minimize the transfer of data back and forth from the host to the device.

One solution to the memory bandwidth problem is to avoid transferring the response images and features back to the CPU by simply evaluating the decision trees natively on the GPU. This solution is a natural one since we can evaluate the classifier over each sliding window in parallel, as well as evaluate each decision tree of our boosted classifier in parallel for each window. The only necessary memory transfer back from the GPU is then the classifier result for each window instead of an entire feature vector, directly translating to an immense bandwidth savings. Thus the choice of a light-weight and parallelizable classifier that can be stored and evaluated easily on the GPU is key to our object detector implementation. This modification yielded a 2x to 3x speedup over simply performing the correlations on the GPU.

In total, offloading our entire detector computation to the GPU has reduced the testing time of our classifiers dramatically. Our software reference implementation (which uses the well-optimized OpenCV library to perform the patch convolutions) requires roughly 5 minutes on a 2.66GHz Xeon workstation to generate all detections for a single object class on a single image. In sharp contrast, our CUDA-based implementation executing on an nVidia 8800GTX can be executed comfortably in under 10 seconds. As seen in Figure 3, this yielded a relative speed boost of nearly 40 times relative to our (software) reference implementation. To demonstrate the value of leveraging GPU hardware, we also show the relative speed boost using a newer nVidia 9800GTX. This GPU is only a single generation beyond the 8800GTX (being released approximately 6 months apart), yet we already see that our detector's running time now improves over the software implementation by more than 90 times on average.

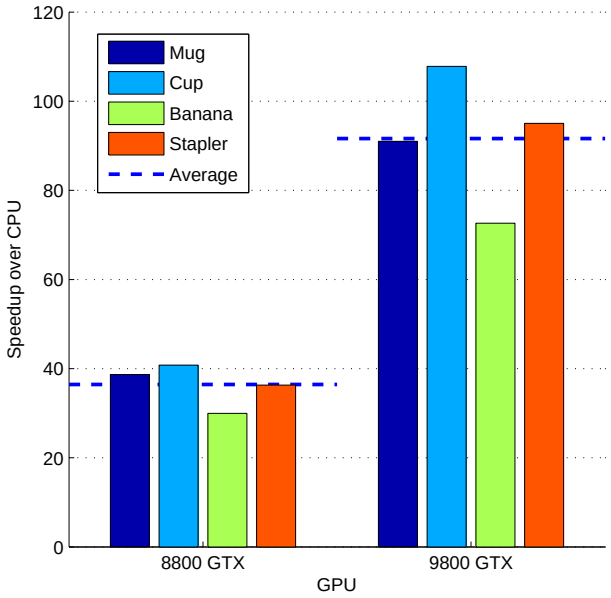

Fig. 3. The factor speedup of our GPU detector implementation relative to our software reference implementation for two nVidia GPUs (from succeeding product families).

The primary explanation for the large speedup is that our dictionary patches are relatively small, allowing the GPU implementation to operate almost entirely on cache memory. In hindsight, the key benefit of the GPU implementation is not simply that GPUs have greater computational throughput (which, at present, is roughly 10x greater overall than high-end CPUs [21]). Instead, we find that because the GPU exposes control of the cache directly to the user, it admits highly optimized implementations of our feature computations that would be much more difficult to implement on CPUs. We believe that focusing on cache-friendly algorithms is thus an important direction for future research, since this is critical for high-speed operation on both GPU and CPU architectures. Indeed, our experience with memory bandwidth limitations on the GPU highlights this conclusion: algorithms that minimize memory access will accelerate profoundly with improvements in processing power.

V. EXPERIMENTS

We demonstrate our entire detection system on imagery of office scenes collected by our robot. As described previously, these images include intensity, gradient, and depth data.

TABLE I
DICTIONARY SIZES AND RUN TIME FOR SINGLE IMAGE

Object	Dict. Size	GPU Time	CPU Time
Mug	590	2.96 s	286 s
Cup	540	3.13 s	320 s
Stapler	472	3.90 s	372 s
Banana	827	4.16 s	302 s

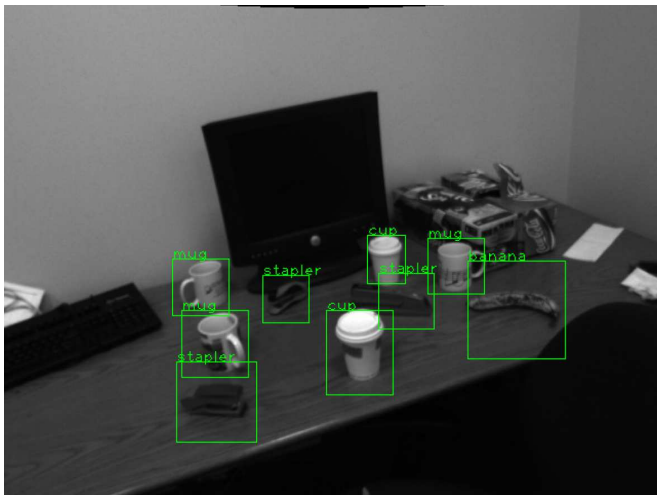

Fig. 4. A typical scene where all objects have been detected and classified correctly by our classifier (best viewed in color).

Fig. 5. A scene where some objects are missed.

TABLE II
OBJECT DETECTION ACCURACY

Object	Count	Hit	False Pos.	Precision	Recall
Mug	67	63	1	0.984	0.940
Cup	43	41	0	1	0.953
Stapler	55	30	0	1	0.54
Banana	21	5	0	1	0.23

Our approach is demonstrated with four different objects: coffee mugs, disposable paper coffee cups, office staplers, and bananas. The first two of these objects have many similar features, making them easy to confuse with one another, while the other two objects are elongated and appear different from various orientations.

For training we acquired 150 images (including depth data) of office scenes consisting of the objects we want to identify. Each image typically contains between one and four instances of an object, each of which is labeled by hand using a bounding rectangle. For each object, this yields a training set consisting of several hundred positive examples. We also train on a background negative set consisting of positive examples from all other object classes as well as a large fraction of all of the examples considered by the sliding window algorithm that do not overlap positively labeled objects. This procedure yields between 5000 and 7000 negative examples for each training image, yielding nearly a million negative training examples for each classifier.

Our decision trees are trained as described in Section III-C. Training for a single object class takes approximately 2 hours for 200 rounds of boosting (using 32 cores on our computing cluster).

We extract 1200 random patches from each set of positives to build a dictionary for the corresponding objects. After training, the dictionary is pruned to retain only features actually used by the decision tree algorithm. Table I reports our average detection times for each object per image along with the final dictionary sizes (GPU time is measured on the 9800 GTX).

To test the accuracy of our detector, we ran it on 20 unseen images from newly imaged office scenes containing instances of the 4 object classes. Examples of two typical scenes are shown in Figure 4 and Figure 5. In Figure 4, our detector correctly identifies all 9 target objects in the scene. Figure 5, however, shows a case where the detector does not do as well. Since our classifiers have been trained to be extremely conservative as a result of the large negative training set, some objects are ignored (incorrectly) as background clutter.

Table II shows our results for all of the objects using a classification threshold that was fixed a priori to probability 0.5. Notice that the most difficult objects are those that exhibit large amounts of variation over different views. This is because the fixed feature patches used by our detectors are not naturally suited to this scenario. In principle, this could be solved by training separate classifiers for each view of the object. Nonetheless, in cases where views of the object do not differ too much (like mugs and cups) the classifier performs extremely well.

VI. CONCLUSION

In this paper, we have demonstrated that we can perform reliable object detection in well under 5 seconds using consumer-grade graphics hardware with a straight-forward learning algorithm that is easy to implement and train. More importantly, however, we have presented a system that is scalable at every point of its execution. From distributed training to the fully GPU-based detection algorithm itself, every step of the pipeline benefits substantially from the predictable upward trends in computing power and data set sizes.

VII. ACKNOWLEDGMENTS

The authors thank Olga Russakovsky for helpful discussions. Adam Coates is supported by a Stanford Graduate Fellowship. Support from the Office of Naval Research under MURI N000140710747 is gratefully acknowledged.

REFERENCES

[1] M. Quigley, S. Batra, S. Gould, E. Klingbeil, Q. V. Le, A. Wellman, and A. Y. Ng, "High-accuracy 3d sensing for mobile manipulation: Improving object detection and door opening," in *IEEE International Conference on Robotics and Automation*, 2009.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April 1965.

[3] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A characterization of ten hidden-surface algorithms," *Computing Surveys*, vol. 6, no. 1, March 1974.

[4] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *CVPR 2008*, June 2008.

[5] A. Torralba, R. Fergus, and W. Freeman, "80 million tiny images: a large dataset for non-parametric object and scene recognition," in *PAMI*, 2007.

[6] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *CVPR*, 2006, pp. 2161–2168.

[7] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *39th Annual Meeting on Association for Computational Linguistics*, 2001.

[8] P. Viola and M. Jones, "Robust real-time object detection," *IJCV*, 2001.

[9] A. Torralba, K. Murphy, and W. Freeman, "Sharing visual features for multiclass and multiview object detection," *PAMI*, 2007.

[10] Y. Luo and R. Duraiswami, "Canny edge detection on nvidia cuda," *Computer Vision and Pattern Recognition Workshops*, pp. 1–8, June 2008.

[11] S. Heymann, K. Mller, A. Smolic, B. Frhlich, and T. Wiegand, "Sift implementation and optimization for general-purpose gpu," in *15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2007.

[12] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2008 (VOC2008) Results," http://www.pascal-network.org/challenges/VOC/voc2008/workshop/index.html.

[13] Y. Lecun, F. J. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *CVPR*, 2004.

[14] S. Gould, P. Baumstarck, M. Quigley, A. Y. Ng, and D. Koller, "Integrating visual and range data for robotic object detection," in *ECCV Workshop on Multi-camera and Multi-modal Sensor Fusion Algorithms and Applications (M2SFA2)*, 2008.

[15] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *CVPR*, 2005.

[16] H. A. Rowley, S. Baluja, and T. Kanade, "Human face detection in visual scenes," in *Advances in Neural Information Processing Systems*, 1995.

[17] V. Ferrari, L. Fevrier, F. Jurie, and C. Schmid, "Groups of adjacent contour segments for object detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2008.

[18] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting," Dept. of Statistics, Stanford University, Tech. Rep., 1998.

[19] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[20] K. Alsabti, S. Ranka, and V. Singh, "CLOUDS: A decision tree classifier for large datasets," in *4th Intl. Conf. on Knowledge Discovery and Data Mining*, 1998.

[21] *nVidia CUDA Programming Guide*, NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050. [Online]. Available: http://developer.download.nvidia.com/