Massively Parallel Multiclass Object Recognition

Helmut Sedding, Ferdinand Deger, Holger Dammertz, Jan Bouecke, Hendrik P. A. Lensch

Ulm University, Germany

Abstract

We present a massively parallel object recognition system based on a cortex-like structure. Due to its nature, this general, biologically motivated system can be parallelized efficiently on recent many-core graphics processing units (GPU). By implementing the entire pipeline on the GPU, by rigorously optimizing memory bandwidth and by minimizing branch divergence, we achieve significant speedup compared to both recent CPU as well as GPU implementations for reasonably sized feature dictionaries. We demonstrate an interactive application even on a less powerful laptop which is able to classify webcam images and to learn novel categories in real time.

Categories and Subject Descriptors (according to ACM CCS): I.4.8 [Image Processing and Computer Vision]: Scene Analysis, Subject: Object recognition

1. Introduction

Biologically motivated object recognition approaches advanced fast in the recent years. Motivated by the speed and accuracy of the primate brain [TFM96], object recognition based on quantitative analysis of visual cortex [RP99] was extended to real world image recognition [SWP05, ML06, SWB*07, ML08]. In contrast to classical computer vision algorithms, these models operate on unknown image classes without requiring parameter changes. Common to all of them is the feed forward processing pipeline which is composed of simple but compute intensive layers [SKC*05, SOP07]. After a couple of transformations the response to a visual feature dictionary is used to classify images. While being relatively simple, these models achieve high recognition rates on real world recognition tasks which are highly complex [PCD08].

The cortex-like structure of biologically motivated object recognition systems is inherently parallel and therefore can be mapped reasonably well onto current graphical processing units (GPUs). The many-core architecture of GPUs and their flexible programming model [OJL*07, OHL*08, NBGS08] allows us to carry out the simple processing tasks of each model layer at a massively parallel scale with multiple thousands of threads in flight. In particular, we address the biologically motivated object recognition system by Mutch&Lowe [ML06]. This shift- and scale invariant object recognition model achieves good accuracies on small dictionaries while not being overly complicated.

Our parallel CUDA implementation follows a couple of general design principles such as implementing the entire pipeline on the GPU, rigorously optimizing the layout of data structures and reusing data between threads to reduce memory bandwidth, and, finally, minimizing branch divergence. It shows that our implementation is vastly faster than a reference CPU implementation, and still significantly faster than a current GPU implementation of the same model inside the Matlab framework by Mutch et al. [MKP10] for reasonable dictionary sizes.

The required time per image is indeed so low that real time applications now are possible. For a reduced feature set which is still sufficient to classify webcam images robustly, we demonstrate an interactive application even on the low-end GPU of a standard mobile computer. The system is able to learn new categories and to classify images on the fly.

1.1. Related Work

The study of the visual cortex in primate brains allowed for a quantitative object recognition model [RP99] which has a strict feed forward structure [SKC*05]. The model was first applied to the recognition of real world images by Serre et al. [SWP05]. They achieved good performance even with a small training set.

Mutch&Lowe extended the model of Serre et al. to hierarchical processing [ML06]. Instead of applying scaled filters,

© The Eurographics Association 2010.

they create an image pyramid of several scales. Further improvements are achieved by sparsification and localization of features.

Several parallel GPU implementations for various cortex-like recognition systems exist today. In contrast to classical computer vision algorithms which are specialized to specific object types, these models allow for very general object recognition tasks. Woodbeck et al. [WRC08] utilize earlier pixel shaders [SAS07] to implement a different biologically inspired model consisting of seven layers.

Chikkerur developed a parallelization for the original model of Serre et al. on CUDA [Chi08]. He reports some speedup, but parallelized only parts of the processing pipeline and kept others on the CPU. They thus give away performance and cannot reach real time performance.

Recently, Mutch et al. developed a more general framework to easily cover various cortical models [MKP10]. They interfaced CUDA from within Matlab to speed up time consuming calculations while keeping an easy to use interface. In the result section we will refer to this work in detail. In contrast, we developed a self-contained application which allows for fast real time processing of images, even on small mobile devices.

2. Feature Extraction Inspired by Visual Cortex

Object recognition is generally divided into two parts, feature extraction with weighting and object classification [SWP05, LBM08]. We built our system closely along the approved base object recognition model of Mutch&Lowe [ML06] which we will describe in the following. In our work we concentrate on the parallelization of the feature extraction and weighting pipeline and use an existing support vector machine implementation for the classification [CL01].

The learning step consists of two phases. The feature extraction phase builds a feature dictionary out of a training image set. The feature weighting phase then measures the similarities of all features to all training images. Both phases use the same processing steps.

The processing pipeline, following the base model of Mutch&Lowe [ML06], is outlined in figure 1. A scale and shift invariant feature vector is computed for input images by two subsequent steps, measuring the response to Gabor filters, and to a set of feature patches, respectively.

These two steps are similar in structure and consist of a simple layer (S) and a complex layer (C) each:

- S-layers compute convolutions with templates or patches. The selectivity for a specific feature is increased by applying a bell-shaped weighting curve to the resulting response.
- C-layers performs non-linear reduction, namely maximum extraction over an area or an entire pyramid followed by sub-sampling. They reduce the amount of information and establish scale and shift invariance.

In the following, we describe the basic principles of each layer.

2.0.1. Image Layer

Images are converted to grayscale. A pyramid is created by down sampling using bicubic interpolation with scale factor $r = 2^{-1/4} \approx 84\%$.

2.0.2. Gabor Filter Layer (S1)

The first layer (S1) measures the response to directional features using Gabor filters. They implement edge detection along a fixed number of orientations. The Gabor filter function $G_\theta(x,y)$ with orientation $\theta \in [0,\pi]$ at a discrete position (x,y) is defined as:

$$G_\theta(x,y) = \exp\left(-\frac{(x_0^2+\gamma^2 y_0^2)}{2\sigma^2}\right)\cos\left(\frac{2\pi}{\lambda}x_0\right) \quad (1)$$

with $x_0 = x\cos\theta + y\sin\theta$ and $y_0 = -x\sin\theta + y\cos\theta$ (Mutch recommends $\gamma = 0.3$, $\sigma = 4.9$ and $\lambda = 5.6$).

The Gabor filter is also frequency selective. However, frequency invariance is achieved by convolving each scale of the image pyramid with a fixed-size kernel. The filter components are normalized by setting the sum of all values to 0 and the sum of squares to 1. By convolving with g orientations, the S1 layer creates g pyramids ($g = 4$ in our case).

2.0.3. Local Invariance Layer (C1)

The C1 layer increases shift and scale invariance over local areas in all S1 pyramids. This is achieved by determining the maximum in 10×10 regions at each scale. As information is lost by this dilation it is sufficient to shift the mask by 5 pixels in each direction which effectively reduces the size of the resulting C1 pyramids by $\frac{1}{5} \times \frac{1}{5}$ for each orientation.

2.0.4. Patch Extraction

During the training phase a set of tiles is extracted from these C1 pyramids as a vocabulary of visual features. Serre proposed a universal feature dictionary [SWB*07] which could be reused for all types of real world imagery, but there exists none yet. Instead, we try to determine a representative set of patches out of the training images by randomly choosing positions and scales inside their C1 pyramids. A patch comprises all orientations though. It is square and varies in size. The idea is that small patches encode shape and larger patches encode texture [ML06]. After extracting the fixed number of patches from all training images, the dictionary remains constant.

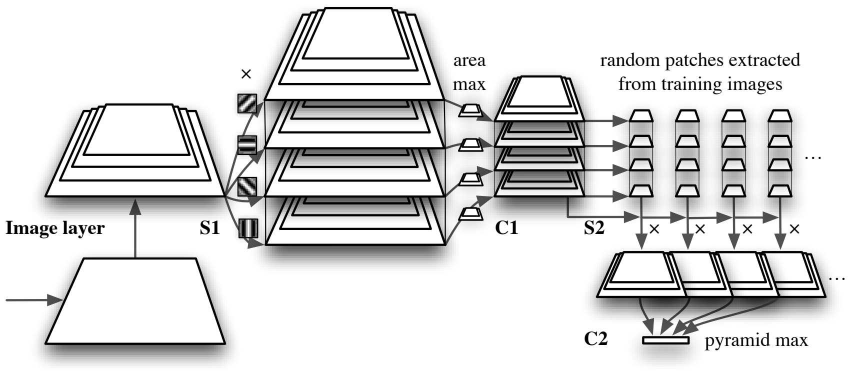

Figure 1: *Hierarchical feature extraction and weighting pipeline.*

2.0.5. Intermediate Feature Layer (S2)

The S2 layer measures the local response of the C1 pyramid of a given image to every patch in the dictionary. A convolution compares the patch to local spatial neighborhoods at all positions and scales, operating on all orientations at once. The response is an euclidean distance value which is tuned with a gaussian radial basis function to increase the relative response for closely matching patches:

$$R(X,P) = \exp\left(-\frac{\|X-P\|^2}{2\sigma^2\alpha}\right) \tag{2}$$

Here, X is a region of the C1-pyramid and P the patch. Both have the same size $n \times n \times g$, where n is the patch edge length and g is the number of Gabor orientations. The standard deviation σ is set to 1. The distance is normalized on patch size with $\alpha = (n/4)^2$. As $R(X,P)$ operates on all orientations at once, orientation information is removed in the result.

2.0.6. Global Invariance Layer (C2)

Finally, the C2 layer removes all position and scale information. Every similarity pyramid is reduced to one single maximum value. The final result is a weighted C2 feature vector containing the maximum responses for all patches.

2.0.7. Classifier

The multiclass classifier is the last step in the object recognition model. For every category, a set of training images is used. The classifier learns using the C2 vectors of the training images. It is advantageous to transform the C2 vectors beforehand [ML06] by adding a bias to set the mean for every patch similarity response to 0 and scaling the standard deviation to 1. A simple linear classifier then determines the best matching category using the C2 vector of the input image. Most commonly used is an all-pairs linear support vector machine (SVM) [CL01], which we also used.

3. Implementation

In this section we describe in detail our design decisions for a high performance implementation of the object recognition pipeline described in the previous section. The first principle in programming for massively parallel architectures is to use many threads that do the same operations, i.e., execute the same kernel on different data. Diverging threads result in a severe performance drop. The second principle is that main memory access is expensive and data that is processed by individual threads needs to be laid out with care for fast throughput. To gain maximum performance, frequently accessed data needs to be copied into fast shared memory that is a scarce resource and accessible only for a smaller set of threads, called a block. Finally, read only access to parts of the main memory can be accelerated using a hardware cache.

The structure of the overall parallelization approach influences also the number of individual kernels and kernel calls that are needed for the final result.

Our framework is implemented in CUDA [NVI09] but the described principles can directly be transferred to other massively parallel programming languages like OpenCL or Microsoft DirectCompute. The code is open source and can be found at http://github.com/cumorc/cumorc.

3.1. Memory Structure

Important design decisions have to be made for the data structures and the organization of the data flow to reduce bandwidth utilization between CPU and GPU memory, as well as on the GPU itself.

3.1.1. Main Memory to GPU Transfer

A significant factor is how much data is transferred between the main memory of the CPU and the GPU. We reduce the amount to a minimum by implementing the entire pipeline outlined in figure 1 on the GPU. One needs to upload the

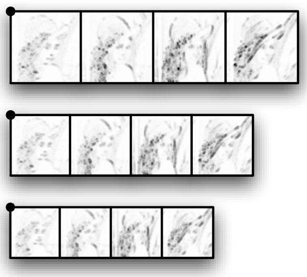

Figure 2: *Data layout for dense storage of one C1 pyramid for four Gabor orientations at three scales.*

input image and the patch dictionary and finally retrieve the compressed C2 vector. Beyond that, no intermediate results will be transferred.

3.1.2. Texture Cached Read-Only Access

The texturing unit of graphics processors provides a cached data access on arrays and allows for automatic boundary handling. This access is read-only but textures can be bound at runtime to GPU arrays written by previous kernels. Texture caches are optimized for 2D data which has to influence the data layout.

3.1.3. Shared Memory

Every GPU processor provides a very fast, but small shared memory. It is specifically suited for data which is accessed multiple times by several threads within a block. Both the S1 and the S2 layer convolve small patches with the image pyramid, we assign one patch to each block and load the patch to shared memory beforehand.

3.1.4. Image Pyramid Data Structure

One of our primary design goals was to allow for flexible image sizes and for arbitrary scaling factors between pyramid levels. We apply texture caching to accelerate access to the image pyramids, our most important data structure. Since there is no standard way of storing image pyramids we evaluated different approaches. Using 1D textures is very flexible and yields basic caching. This already improves the performance by 10% over simple global memory storage.

Memory efficient storage of pyramids using 2D textures requires to create an extra texture for each pyramid scale. Managing several texture references is currently tedious and inflexible in CUDA. This problem can be alleviated by using only a single texture reference and using multiple kernel calls that each process a single scale. Even though calling a kernel imposes some overhead this is about 50% faster than multiple textures in one kernel.

Filtering with g Gabor orientations in layer S1 generates multiple image pyramids of the same size. Since a single texture reference per kernel is fastest we group each scale of the different pyramids into a single texture. This is illustrated in figure 2 for 4 C1 pyramids with 3 scales. Grouping multiple images in a single texture enables us to process a single layer of multiple image pyramids within a single kernel call but an additional boundary treatment is needed.

We also use this data structure for storing all patches as it scales very well and allows for varying numbers of elements per scale.

3.2. Layer Parallelization

We paid special attention to the parallelization of each layer, and focus in this section on the main principles when implementing a cortex-like model. For GPUs, concurrent computation (lockstep) only happens if the threads within a block share the same codepaths. For high performance it is mandatory to select a data parallel algorithm [HS86] where the data might be different but where each thread executes the same operations including branching decisions. A second issue is minimizing bandwidth.

3.2.1. Single Thread per Output (Image Layer, S1, C1)

The first layer (Image Layer) uses the source image to create an image pyramid. We create one thread for every output pixel and each thread then reads a small area to interpolate over, effectively exploiting the caching hardware.

Creating one thread per output pixel is a principle which is directly reused in the S1 and C1 layers. It actually corresponds to a neuron which computes its output by weighting inputs at several dendrites.

3.2.2. Merging Layers S2 and C2

The most time critical layer is S2. A convolution computes the similarity between the p patches and the C1 pyramid of the image and produces p similarity maps. The C2 layer then extracts the maximum of each similarity map.

We partition the problem such that each block of threads operates on a single patch and performs the convolution with one scale of the pyramid at all orientations. Each thread again is responsible for one output pixel. As we fix the number of threads n_t in the block, it might be necessary to iterate multiple times within the kernel to compute all output pixels. We need p blocks, and the algorithm scales with the number of processing units.

This partitioning carries a number of benefits. Blocks can run concurrently even if they do not execute the same codepath. Therefore, each block can easily handle a different patch size. As every thread in the block needs access to the same patch, we need to load the patch data into fast shared memory only once. 2D texture caching is used to access the C1 pyramid.

An additional, significant benefit is achieved by integrating the S2 and C2 layer into the same kernel. S2 creates p similarity pyramids, while the C2 layer reduces every pyramid to one maximum value. Each block thus needs to return a single value. We completely avoid waisting any memory or bandwidth for the similarity pyramid by extracting the maxima already when convolving with the patch. When scanning over the image, each of the n_t threads only keeps track of one maximum value. A parallel reduction algorithm then computes the maximum of the n_t values within a block in $\log n_t$ steps [HS86, HSO07]. The reduction can operate very fast because the threads of one block can still access shared memory. Our implementation guarantees a close to minimum bandwidth requirement to global memory.

4. Results

4.1. Experiments with Caltech 101 database

For evaluation, we use the entire Caltech 101 image database [FFFP04] which provides a set of real world photographs. It consists of 9144 pictures in 101 object categories and one background category. 15 training images are randomly selected out of each category which is sufficient for the applied classification framework. For all benchmarks, we measure the random patch extraction, the C2 vector calculation by a new set of training pictures and independently the evaluation of the accuracy with at most 100 different images. We exclude the time of the SVM but apply the same SVM for all tests. We took care that all tests used the same random numbers and thus the same set of images.

We compare our system with both the CPU reference implementation (FHLib) of Mutch&Lowe [ML06] and their recently published GPU cortex simulator framework (CNS) [MKP10] which runs in Matlab. Both are publicly available. We use the same set of parameters as documented for FHLib and applied them also to the CNS framework. A direct comparison is difficult though as the FHLib allows the shorter edge length to be 140 pixel at maximum while CNS limits the longer one. As our implementation allows for arbitrary images sizes we compare to both implementations separately.

Table 1 analyzes the classification performance. For the base setup with 4075 patches and shorter edge lengths of 140, we acheive a classification rate of 37%, averaging over 8 runs. This is a bit better than Lowe's&Mutch's 33% (base model).

To demonstrate that the classification rate behaves similarly, we show the performance at different patch numbers in Figure 3. The accuracy grows logarithmically and flattens out at more than 1500 patches. Both implementations show the same behavior here, indicating that a certain dictionary size is sufficient to robustly differentiate image classes. We also compare the classification rate at different source image sizes in the bottom row of Figure 3.

Table 1: *Classification rates on the Caltech 101 dataset with varying number of training images per category (i./cat.). The results are averaged over 8 independent runs. The performance of CNS is not directly comparable as it omits the data transformation before calling the SVM.*

Model	Impl.	15 i./cat.	30 i./cat.
Serre et al. [SWP05]	CPU	35%	42%
Mutch&Lowe Base Model (FHLib) [ML06]	CPU	33%	41%
Mutch et al. Base Model (CNS-FHLib) [MKP10]	GPU	29%	37%
Our implementation	GPU	37%	46%

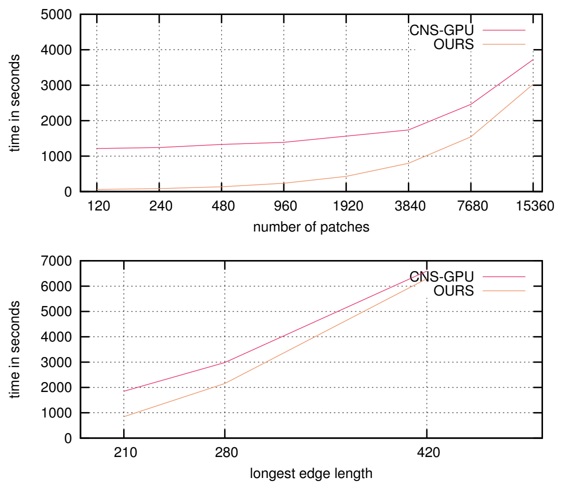

Figure 4: *Our systems competes well against the recent GPU implementation of the CNS-FHLib (CNS-GPU) on the Caltech 101 database. Both, in the number of patches, as well as the different sizes, our system is faster. The big initial gap might be explained by some overhead, that comes along with the Matlab Framework CNS-GPU uses.*

Figure 4 shows the performance benchmark against the recently published GPU CNS-FHLIB [MKP10]. In the test-range from 120 to 15360 patches and longest edge length from 210 to 420 we consistently outperform this implementation. For 4075 patches, processing for one image takes 86.4 ms on a NVIDIA GeForce285GTX, and only 8.9 ms for 240 patches.

4.2. Mobile Real Time Application

The significant speedup, especially for moderate numbers of patches, allows us to address real-time object recognition even on small mobile GPUs such as a low end NVIDIA Geforce 9400M in an Apple Macbook Pro.

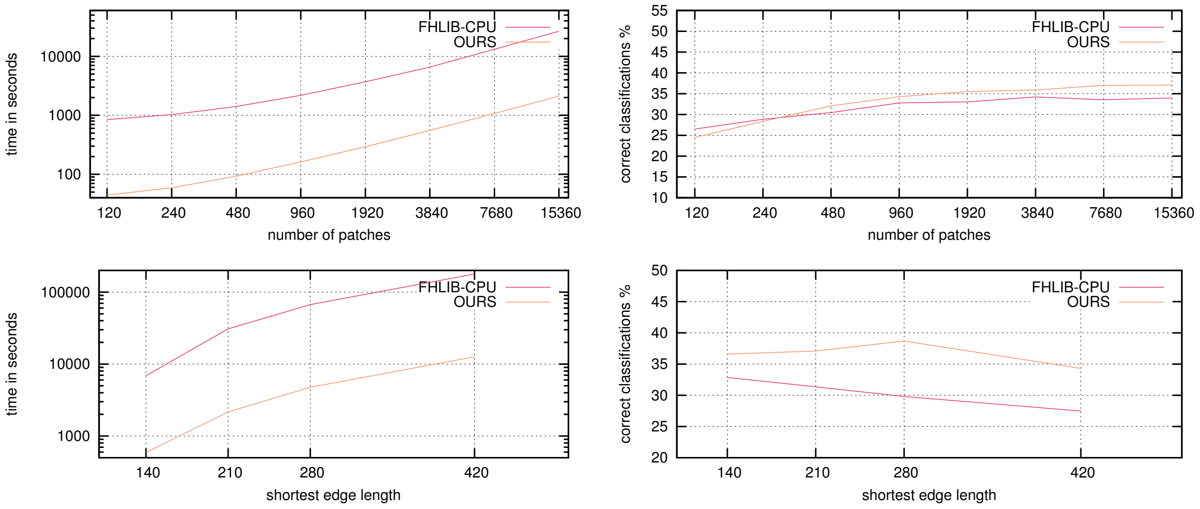

Figure 3: *The plots in the top row show a comparison of the accuracy and performance for a varying number of patches for FHLib and our GPU-based implementation on 9144 images. The pictures are resized to 140 px (shorter edge length). We achieve the same accuracy as the reference at a more than 10 fold speedup.*
The bottom row show the performance and accuracy test between our system and the FHLib for different image sizes. For this experiment the Caltech Library is interpolated bicubicly and resized. Due to the quadratic growth of the workload both implementations show the same increase in their runtime performance. The accuracy plot shows that both implementations stay in approximately the same range.

As one test setting we used the COIL-100 [NNM96] data set as it contains representative images for a webcam application with subtracted background [Pic04]. We obtain good classification performance even for small dictionaries (82% with 30 patches, 93% with 120 patches measured on the COIL-100 data set - see Figure 5). We implemented a live test setting using the built-in webcam. The application learns and queries pictures within a tenth of a second. At webcam resolution (640 × 480 scaled to 186 × 140) and using 480 patches the feature extraction of an image is performed within 452 ms, with 120 patches even in under 100 ms. As we cannot generate a feature dictionary from the live data stream we use a static feature dictionary as proposed by Serre et al. [SWB*07]. To test the suitability of our current dictionaries, we classified he COIL-100 data set with Caltech 101 patches. We thereby achieved almost the same classification rate as by extracting the patches from the same dataset, as Figure 5 proves. For our real-time application we capture several images and calculate the SVM model in under a second. This is then used to classify images in realtime. You can see our application in Figure 5 and in the supplemental material.

5. Conclusion

Biologically motivated image recognition models lend themselves to be implemented on massively parallel platforms such as current many-core GPUs. By considering a set of parallel programming principles we were able to achieve a significant speedup, even compared to recent GPU implementations. We demonstrate real time image classification on a low end GPU. Availability of low power many-core processors such as NVIDIA Tegra could bring real time object recognition even to handheld devices.

A number of improvements could be added to the current framework, e.g., [ML06], to further improve classification rates. With a small but universal feature dictionary we could provide online learning capabilities. In addition, we plan to apply the proposed design decisions to other biologically inspired algorithms.

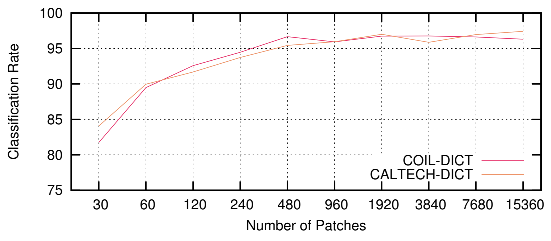

Figure 5: *The plot shows COIL-100 classification rates with a feature dictionary generated out of test images (COIL-DICT) and with a dictionary generated from the Caltech 101 dataset (CALTECH-DICT).*

References

[Chi08] CHIKKERUR S.: CUDA implementation of a biologically inspired object recognition system. *MIT online* (2008).

[CL01] CHANG C.-C., LIN C.-J.: *LIBSVM: a library for support vector machines*, 2001.

[FFFP04] FEI-FEI L., FERGUS R., PERONA P.: Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *CVPRW '04: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04) Volume 12* (2004), p. 178.

[HS86] HILLIS W. D., STEELE JR. G. L.: Data parallel algorithms. *Commun. ACM 29*, 12 (1986), 1170–1183.

[HSO07] HARRIS M., SENGUPTA S., OWENS J.: Parallel prefix sum (scan) with CUDA. *GPU Gems 3*, 39 (2007), 851–876.

[LBM08] LABUSCH K., BARTH E., MARTINETZ T.: Simple method for high-performance digit recognition based on sparse coding. *behaviour 14* (2008), 15.

[MKP10] MUTCH J., KNOBLICH U., POGGIO T.: CNS: a GPU-based framework for simulating cortically-organized networks. *MIT-CSAIL-TR-2010-013* (2010).

[ML06] MUTCH J., LOWE D. G.: Multiclass object recognition with sparse, localized features. In *CVPR (1)* (2006), IEEE Computer Society, pp. 11–18.

[ML08] MUTCH J., LOWE D. G.: Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision 80*, 1 (2008), 45–57.

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with CUDA. *Queue 6*, 2 (2008), 40–53.

[NNM96] NENE S., NAYAR S. K., MURASE H.: *Columbia Object Image Library (COIL-100)*. Tech. rep., CUCS-006-96, 1996.

[NVI09] NVIDIA: CUDA programming guide 2.3.

[OHL*08] OWENS J., HOUSTON M., LUEBKE D., GREEN S., STONE J., PHILLIPS J.: GPU Computing. *Proceedings of the IEEE 96*, 5 (2008), 879–899.

[OJL*07] OWENS, JOHN D., LUEBKE, DAVID, GOVINDARAJU, NAGA, HARRIS, MARK, KRUGER, JENS, LEFOHN, AARON E., PURCELL, TIMOTHY J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (March 2007), 80–113.

[PCD08] PINTO N., COX D. D., DICARLO J. J.: Why is real-world visual object recognition hard? *PLoS Comput Biol 4*, 1 (01 2008), e27.

[Pic04] PICCARDI M.: Background subtraction techniques: a review. vol. 4, pp. 3099 – 3104 vol.4.

[RP99] RIESENHUBER M., POGGIO T.: Hierarchical models of object recognition in cortex. *Nature Neuroscience 2*, 11 (1999), 1019–1025.

[SAS07] SERRA O., ALQUÉZAR R., SANFELIU A.: Parallelization of Bio-Inspired Convolutional Networks for Object Recognition Using the GPU. *IRI-DT 2007/04* (2007).

[SKC*05] SERRE T., KOUH M., CADIEU C., KNOBLICH U., KREIMAN G., POGGIO T., SERRE T., KOUH M., CADIEU C., KNOBLICH U., KREIMAN G., POGGIO T.: A theory of object recognition: Computations and circuits in the feedforward path of the ventral stream in primate visual cortex. In *AI Memo* (2005).

[SOP07] SERRE T., OLIVA A., POGGIO T.: A feedforward architecture accounts for rapid categorization. *Proceedings of the National Academy of Sciences 104*, 15 (2007), 6424.

[SWB*07] SERRE T., WOLF L., BILESCHI S., RIESENHUBER M., POGGIO T.: Robust object recognition with cortex-like mechanisms. *IEEE Transactions on Pattern Analysis and Machine Intelligence 29* (2007), 411–426.

[SWP05] SERRE T., WOLF L., POGGIO T.: Object recognition with features inspired by visual cortex. In *In CVPR* (2005), pp. 994–1000.

[TFM96] THORPE S., FIZE D., MARLOT C.: Speed of processing in the human visual system. *Nature 381*, 6582 (June 1996), 520–522.

[WRC08] WOODBECK K., ROTH G., CHEN H.: Visual cortex on the gpu: Biologically inspired classifier and feature descriptor for rapid recognition. pp. 1 –8.