VISYM: A Platform for Large Scale Computer Vision

Jeffrey Byrne, Jack Sim and Jianbo Shi
University of Pennsylvania
{jebyrne, jiwoong, jshi}@cis.upenn.edu

Abstract

What if every vision algorithm ever written was ready to run via browser, on images from any network connected camera? In this paper, we introduce visym.com, a new platform to easily share and distribute computer vision as a real time web service. We believe that this platform can have a broad impact for the vision research community. Specifically, it can provide a means for seamless MATLAB algorithm and data sharing, transparent coarse grained parallelization, intermediate results reuse, and resource pooling, all without changing current R&D software practices. Furthermore, exposing vision as a real time web service opens up development to the web, to address the long tail of vision applications. A beta test of this platform is currently underway at http://beta.visym.com.

1. Introduction

Modern computer vision research and development has exploded since its origins in the 1950s to include a wide variety of applications. For example, a scan of research archives including CVPR, ICCV and PAMI shows over 107,000 papers with 7300 keywords associated with computer vision, with algorithms and applications ranging from abnormal event detection to zip code recognition. In general, these applications following a long tailed distribution, such that there are some popular applications such as face recognition, but many more niche applications. It can be argued that the “killer app” for vision is not a single popular application, rather it is a platform for distribution of niche apps.

The challenge of distribution is that vision is a big data problem. Recent work in internet scale vision has moved to databases with millions of labeled training images [1, 2, 3], and this number will only increase [7]. Simply sharing terabyte scale databases is a challenge due to limited bandwidth, which means that practical sharing of datasets requires physically shipping hard drives between groups. Furthermore, sharing functions such as data driven classifiers trained on big datasets often have parameters which require sharing the majority of the dataset as well [2, 5, 12]. Also, processing big datasets requires a large computational platform. Most vision algorithms exhibit coarse grained parallelism, which can be exploited for large scale processing. However, setting up a cluster to handle peak load may not be cost effective for individual research groups, and these computational resources still need datasets close by. So, there is a need for a large scale vision platform to share big data with the computational resources to process this data by exploiting data locality.

Vision is also a big, diverse data problem. Complex images may require many functions operating in parallel on a single image to extract all relevant content [9]. Furthermore, many algorithms share similar preprocessing and feature extraction components, and these components are constantly evolving due to the dynamic nature of vision software research. So, there is a need for a big library of diverse vision services to address complex images, and to allow components to be seamlessly updated as necessary.

Finally, vision would benefit from more reproducible research. There are of course many reasons why an author would not provide source code to accompany a paper, however common issues include: lack of incentive to main-
tain open source, embarrassment about sharing quick and dirty code, IP or ITAR restrictions and headaches of packaging and regression testing. Also, even when a MATLAB toolbox is available, it can be difficult to use due to MATLAB version incompatibilities, licensed toolbox requirements and library dependencies. Since MATLAB is the dominant language in vision, there is a need to enable researchers to quickly and easily share MATLAB code without exposing source, to allow colleagues to reproduce results and test on new images. Furthermore, making such functions available would allow non-vision developers to test and evaluate, opening development to the broader community on the web.

In this paper, we introduce the Visym platform to the research community to address these observations. The Visym platform is:

1. Server appliance: We provide a software appliance, a virtual machine image containing a light web server, database, and web service handler supporting MATLAB services. This appliance is the core building block of the Visym platform. Appliances can be provisioned to run on standard hypervisors in any infrastructure-as-a-service provider or deployed in a user’s local cluster, to provide large scale distributed processing. Every appliance has access to all data and services available on the platform through a sharded URI to URL naming scheme, and every appliance exposes a web service API for web based service calls and low latency data access.

2. Web service API: A web based programming interface that provides a language agnostic method of remote procedure calls of vision services. The API is consumable by any language, and is accessible from any appliance via browser or mobile devices.

3. MATLAB client tools: We provide an installable MATLAB toolbox which abstracts the API to native MATLAB objects. Calling vision services on the platform is simply programming in MATLAB, and building new vision services from existing MATLAB code is a one-click operation. All API calls are asynchronous, and they provides transparent parallelization based on referentially transparent services, lazy evaluation of results, and strict consistency from MATLAB variable causality. MATLAB clients get coarse grained parallelization for free.

In this paper, we describe the architectural design of the server appliance and API, then describe the MATLAB client tools from a user perspective. Next, we evaluate the transparent parallelization performance of three common vision toolboxes built using the MATLAB client tools, executed using multiple server appliances deployed in an operational cloud. This demonstrates scalability and usability to justify large scale deployment.

2. Related Work

In the early 90s, DARPA introduced the Image Understanding Environment (IUE), a five year research program to provide a common development environment for vision [6]. The goal was to increase research productivity and technology transfer by enabling sharing and reuse of functions and intermediate results. This goal was not fully realized, however there were two primary lessons learned of relevance to this effort. First, the IUE provided an API defining common data structures, however this centralized process was carried out in closed workshops and did not address the needs of the long tail. Second, IUE forced researchers to use C++ and Lisp, so those who develop in other languages were required to port their code, a time consuming process with few research incentives. The lessons learned are that a vision platform must be language agnostic, and the API development must be open to the community.

Internet Vision is a growing research area, where one long term goal is the Visipedia [9], which highlights the technical challenges for vision on the web. This platform could provide the computational engine and web services for the automatata in such a community driven effort. Related efforts in large scale labeled image datasets include LabelMe [1], ImageNet [2], Lotus Hill [3], where total images have exceeded eleven million. However, even with these dataset sizes, analysis shows that larger datasets are needed [7]. Recent examples of large scale vision that would benefit from a vision platform include scene parsing [10], scene understanding [11], object categorization [12], localization [5], structure from motion [13], landmark classification [14].

A platform for large scale vision is an example of data intensive, high performance computing (HPC). Condor is a popular workload management system for grid computing, however Condor requires distributed computing expertise for deployment and maintenance, and Condor job submission tools can be unintuitive for non-experts. Hadoop provides tools for MapReduce based batch processing of large datasets, however these tools are optimized for high throughput over low latency of data access due to the assumptions of the hadoop distributed file system containing huge files split into blocks. Vision data has a natural atomic size (e.g. “the image”), which does not require the overhead of a full distributed file system, and also web deployment requires low latency data access. Microsoft Dryad [8] provides coarse grained parallelism for jobs with sparse dependencies between subtasks, rather than embarrassingly parallel operations expressible in MapReduce or Condor. However, this requires the user to provide a directed acyclic graph expressing job causality constraints for scheduling.

Platform-as-a-service (PaaS) is a commercial offering which provides a platform for distribution of custom de-
signed webapps. Platforms such as Google AppEngine, Microsoft Windows Azure, Joyent, Heroku and Engineyard allow users to deploy Python, Java or Ruby on Rails webapps without worrying about scalability. Infrastructure-as-a-Service (IaaS) providers such as Amazon, Rackspace, or GoGrid provide utility computing and distributed storage rented on demand to customers. This paper can be considered a PaaS provider for MATLAB apps, leveraging IaaS for scalability.

Finally, MathWorks provides tools for MATLAB web deployment and distributed computing. The MATLAB compiler and MATLAB builder NE/JA provides tools for building and deploying MATLAB applications against the MATLAB compiler runtime (MCR) royalty free. However these solutions are not language agnostic, require ≥200MB MCR installation and are synchronous, limiting client parallelization. MATLAB Parallel Computing and Distributed Computing toolbox provide fine and coarse grained parallelism, however they require license fees which limit scalability. This effort is most closely aligned with MATLAB*P [21] for transparent parallelization through operator overloading.

3. Platform Architecture

The design goals of the platform architecture were defined to address the observed need in the research community as outlined in section 1. Specifically, these goals are: scalable, decentralized, interactive, sharable and user friendly. Scalable and decentralized operation, as well as other distributed systems goals such as fault tolerance, load balancing and scheduling are well known and achievable goals in the distributed system literature, and will not be addressed in detail here.

The primary challenges that are unique to this platform are client interactivity and sharing. As discussed in section 2, batch processing tools do exist to create massively data parallel pipelines. However, while monitoring tools such as Flume and Hue allow a user to submit batch jobs and monitor status of Hadoop clusters, such tools do not tightly integrate with the developer’s design workflow. We believe that vision researchers want to write in familiar languages like MATLAB and they want to monitor the intermediate results as MATLAB variables to quickly view the results during evaluation. This is possible since vision datasets and results are typically characterized by a large number of megabyte scale files (images, metadata and parameters), rather than a small number of terabyte scale files common in batch jobs. Furthermore, client interactivity includes web clients, and web interactivity is not a use case for batch processing. Second, since vision datasets are characterized by a large number of files and more importantly a large number of functions for data analytics, our goal is to enable sharing. A typical batch job assumes that input data is manually provided as input to the job creation script, and analytics are custom designed by the batch submitter. In contrast, complex images require many functions that are developed by other researchers, and many share intermediate results from previous executions. Vision researchers should have a platform optimized for image data analytics, so our goal is to make it easy to share and discover functions and data created by others on the platform.

The platform architecture is shown in figure 2. The Visym platform is a set of server appliances deployed in hypervisors at visym.com, IaaS providers and at partner locations. Visym client tools allow MATLAB and Python users to easily connect to the visym platform, and to access any URI deployed on the platform.

A Visym server appliance is a virtual machine image that includes: a light weight HTTP server for static content, a FastCGI server for dynamic content, a Python based web service handler, a non-relational database, and a secure chroot jailed MATLAB process server containing the MATLAB compiler runtime (MCR). The MATLAB server spawns K processes, each with an independent MCR, to take advantage of multcore servers. Each MATLAB process executes shared object libraries built using the MATLAB compiler, reusing the MCR initialization to avoid significant startup overhead. These appliances handle API calls to run MATLAB services, where inputs and function objects are fetched by URI, and results are cached locally by URI for reuse. This API is described in section 3.1.

Uniform Resource Identifiers (URI) are used to provide identification and versioning to all data and functions on the platform. Appliance databases are shared by URI, to horizontally partition and load balance across appliances. The platform API provides name server functionality, to forward URI requests to primary or secondary appliances, which respond with a URI for resource fetch. The URI scheme is described in section 3.2.

The Visym client tools enable users to natively connect to the Visym platform API. Currently, only MATLAB clients are supported, however Python support is in development. The MATLAB client tools abstract the serialization of native datatypes into uploaded MAT or HDF5 files, generate URLs for remote procedure calls, and handle unserialization of results back to MATLAB datatypes. The client tools provide support for monitoring service calls, throwing exceptions and displaying service logs. Furthermore, the client tools allow users to create new services from their MATLAB code. The MATLAB client tools are described in section 3.3 and building new services in section 3.4.

Consider the following use case. Researcher AL writes a new paper, and has a MATLAB function (f1.m) used for evaluation and a set of support functions (f2.m, f3.m, fk.mex). AL installs the Visym MATLAB client tools and runs visym.build(‘f1.m’), which locates all dependencies, builds a service for f1.m and returns a service URI F.
searcher BE wants to test AL’s code. Instead of sharing a toolbox on his website, AL shares URI F with BE. BE installs the client tools and runs \( f = \text{visym.function}(F) \), which returns a function handle \( f \). BE can process \( N \) of her images in parallel using AL’s function:

```matlab
>> for k=1:N
>>     myoutput{k} = f(myimage{k})
>> end
```

This loop will return immediately, and the cell array myoutput stores ticket variables. Each ticket has a URI referencing the result of AL’s function on the platform. These variables overload all MATLAB operators, and can be used natively just like any other variable. When the variable is referenced, the result is downloaded (or blocked until the result is available) and cached. The client tools make service calls easy, and the platform handles the parallelization by scheduling API calls to available appliances, all transparently to BE.

### 3.1. Web Services

The Visym platform includes a REST-RPC hybrid web service API to provide vision-as-a-service functionality. RESTful web services define an API using standard HTTP operations (GET, POST, PUT, DELETE) on URL identified resources, while remote procedure call (RPC) web services define unique functions and arguments posted to a service endpoint. In this platform, a simple helloworld service call is expressed by an HTTP GET to `http://beta.visym.com/api?f=:\text{func:test.helloworld}&args=["me"]`. The arguments in the querystring (following `?`) define a built-in function URI `:func:test.helloworld` identifying the function to execute, called with a javascript object notation (JSON) encoded argument list (`['me']`). Simple vision examples include KLT features [15], and minimum spanning tree segmentation [16] available at beta.visym.com/demo.

Features of this web service include: (i) asynchronous, responses where service calls return immediately with a JSON encoded transaction URI response or ticket. Tickets can be fetched to check status, throw exceptions and show MATLAB command window output of a long running service call, (ii) session based authentication, (iii) JSON serialization to enable JSONP cross domain web services which enable web mashups, (iv) argument formatted as URI, URL or JSON encoded literals and (v) service controls for email notification and response formatting by adding keywords to the querystring.

### 3.2. URIs

Effective sharing of vision data means providing a unique ID to identify images, intermediate results and functions. These IDs should be easy for humans to remember, assigned in a decentralized manner without conflicts, easily versionable, persistent, and should provide a unique ID for every pixel in every image. In this section, we introduce a variation of the tag Uniform Resource Identifier (tag-URI) [17] to achieve these design goals.

Visym URIs are defined by the following general syntax:

```
:owner:tagname-tagversion:type:id#fragment
```

where the full ABNF syntax specification is provided in [17]. The owner is a well formed email address, the tagname-tagversion defines a set label and option dotted decimal version number, the type defines the data type `[img, func, data, txn, var]` for images, functions, data, transactions or variables. The id identifies an element in the set (e.g. image in a dataset), and the fragment identifies a subset of elements in the id (e.g. region of interest in an image).

Visym URIs can be vectorized by dropping the id. For example, `:me@an.edu:myset-1.2:img:` is a well formed URI which refers to the set of all images tagged myset-1.2. Furthermore, for well known datasets, the owner can imply `:myset-1.2:img:` This is a concise means to reference large, user defined datasets.

Finally, Visym URIs are immutable. All URIs have write once, read many access model. This facilitates data coherency given concurrent access, and also provides versioned “snapshots” of data and functions that never change.
3.3. Matlab Client Tools

We provide MATLAB client tools to seamlessly interface with the Visym service. These MATLAB tools are installable from the MATLAB prompt:

```matlab
>> eval(urlread('http://beta.visym.com/installer.m'));
```

which will guide the user through the installation and setup process. The MATLAB client tools provide the following features deployed as a package of classes: service function calls, user management, configuration file management, operator overloading, data management for data upload/download, exception handling and service building.

The client tools were designed to be MATLAB native and seamless for MATLAB users. Calling a Visym service function looks to the end user just like calling a local MATLAB function with MATLAB variables. For example, a function call to compute normalized cuts:

```matlab
>> f_ncuts = visym.function(':func:ncuts');
>> C = f_ncuts(img, 10);
>> imagesc(C());
```

visym.function is a method provided by the client tools for remote service calls. Visym.function is passed a URI identifying a service function (a built-in URI :func:ncuts) and returns a function handle (f_ncuts). When this function handle is called, the client tools serialize and upload the user supplied image variable (img) to the platform, then compute multiscale ncuts on the platform with 10 segments. The function call is asynchronous, and returns immediately, such that output argument 'C' encodes a transaction URI. The user can monitor the status of the function execution (C.state()), and seamlessly download results for user display when execution completes.

The same function can be called using an existing image URI:

```matlab
>> C = f_ncuts(':img:tiger', 15)
```

would perform ncuts on a platform built-in image URI (:img:tiger), which does not require upload of the full image prior to execution. Furthermore, output arguments can be shared and reused. For example, suppose there exists an second function g which uses the output C as an input. The client tools exhibit lazy evaluation, where variables are not downloaded until contents are accessed. When g(C) is called, the client tools include the data URI for C in the service call rather than downloading the intermediate result. However, if C is accessed (e.g., region of interest C(1:10,1:10)), then the variable is downloaded and cached locally as expected.

3.4. Building MATLAB Services

Finally, the visym.build function provides a means for users to build their own services. With one click, the tools parse a directory to determine all M and MEX source dependencies, warn the user about incompatible functions, and upload the package to the platform for building. When complete, the platform emails the user with a function URI for use by visym.function and to share this function with others.

Building services requires that functions are referentially transparent. Referential transparency means that a function has no “side effects” and can be replaced by its output arguments without changing the program. This approach has been used in functional programming to enable caching and transparent coarse grained parallelism. Referential transparency is enforced during building by parsing dependencies for disallowed keywords that change global state (e.g. global variables, file writes, system calls) and prompt the user for modification. MEX files are exempt from this parse, and are recompiled on the server assuming they are referentially transparent.

The resulting MATLAB services are called using visym.function using the returned function URI or could be directly called via REST-RPC API. The semantics of the input and output arguments are defined by the user, which leaves the specific API for sharing vision functions up to the community.

4. Experimental Results

In this section, we evaluate the parallelization performance and service overhead for the platform. We chose three commonly used MATLAB toolboxes: multiscale normalized cuts [18], multiscale global Pb [19] and part based object detection [20]. These toolboxes were chosen due to their wide citation, and since they have nontrivial runtime per image and would benefit from parallelization and reuse.
We were able to build a service for these toolboxes from each of their latest releases without making any modifications to the source code.

The experimental evaluation included the Visym server appliance deployed on four 512MB provisioned slices in Rackspace Hosting. Each appliance was running K=1 MATLAB processes, and one appliance was manually elected to provide simple round robin scheduling of API calls. A remote MATLAB user was running the client tools to build services for interface functions in each toolbox, and to upload 200 Berkeley segmentation training set images to the appliances. The resulting image and function URIs were used to process N images on between one and four servers, with a 2 sec poll interval to check completion status of each service call. Analysis was limited to four servers simply to demonstrate the scalability trend.

Figure 3 shows the total runtime for each algorithm using 1, 2 or 4 appliances. This shows that the runtime performance is inversely proportional to the number of servers, since a 512MB slice from Rackspace provides at most one virtual core. This demonstrates transparent parallelization for an embarrassingly parallel input. Figure 4 shows the overhead statistics for this experiment. The average overhead for the service is 0.267 second per call. This overhead includes time for downloading results of algorithms to local client. We assumed a scenario where there is no upload required except parameters where all input images are pre-uploaded.

Next, we compared the performance to existing parallelization tools using Condor. We installed Condor on all appliance servers, compiled NCuts to a standalone executable using the MATLAB compiler, and wrote a Condor job submission script to replicate service execution. Results show that runtime performance is 1140.3 s, 583.3 s, and 292.1 s for 1, 2 and 4 server execution, nearly identical to platform performance in Figure 3. Standalone executables reload the MATLAB MCR on every execution, and the mean reload time is 1.4 sec. This is approximately equal to the 2 second polling interval for service execution. This shows that Visym platform is equivalent in performance to traditional batch execution, rather than parallelization with sparse dependencies among components which characterize the majority of vision algorithms. This parallelization is a natural extension of the results shown, however evaluation will be future work.

<table>
<thead>
<tr>
<th>Number of Servers</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (s)</td>
<td>1140.3</td>
<td>583.3</td>
<td>292.1</td>
</tr>
<tr>
<td>NCuts</td>
<td>1140.3</td>
<td>583.3</td>
<td>292.1</td>
</tr>
<tr>
<td>Pb</td>
<td>1057.6</td>
<td>537.8</td>
<td>259.4</td>
</tr>
<tr>
<td>Obj Detect</td>
<td>1607.7</td>
<td>777.3</td>
<td>390.7</td>
</tr>
</tbody>
</table>

Figure 4. Service Overhead

4.1. MATLAB Demo

Figure 5 shows a demonstration of the Matlab client tools included in the visym installation. This provides a walkthrough of the primary feature of the Matlab client tools and how they are used for platform interactivity and sharing. This figure is also available in a larger format at http://beta.visym.com/overview.

4.2. Web Demo

Finally, we provide a demonstration of calling vision services from javascript in browser. This demonstration is available at http://beta.visym.com/demo and allows the user to call specific functions on images from any URL and view results. This interface can also be used for evaluation of new MATLAB services, where the image result displayed is the contents of the last figure plotted by the MATLAB service. This provides a built-in browser demo for any vision service for free. The supplementary material shows a video of this demo and usage of the client tools.

5. Future Work

In this paper, we have described a platform for large scale vision and evaluated common use cases. However, there are a number of remaining challenges. We described the server appliance and end user tools of relevance to the vision community, but only briefly touched on distributed systems issues: scheduling, fault tolerance, load balancing, data archival, data consistency, synchronization, replication and security. We believe that these issues can be cleanly addressed within the proposed framework, however such issues will be explored further during the open beta. Furthermore, we addressed embarrassingly image parallel jobs, rather than parallelization with sparse dependencies among components which characterize the majority of vision algorithms. This parallelization is a natural extension of the results shown, however evaluation will be future work.

References


Figure 5. MATLAB client tools demonstration script

% Demonstration of the vispm service

% Copyright 2000 Vispm
% License GPL: <https://www.gnu.org/licenses/gpl.html>
% See doc/wh-70 300-09-09 03/14/02

% User demo

fprintf('This demo is designed to be stepped through using the matlab debugger.\n');
fprintf('New file to upload:\n');

% Currently installed version

vispm.version();

% Enable verbose debugging output. This will describe in detail the
% network transfers that occur for each service call. You will usually
% want this to be turned off.

vispm.debug(true);

% Upload an image file to service (you may be prompted to login)
% However, passwords are not stored locally or in the matlab history
% so use upload('peppers.png')

% Notice that the response from the upload is a 'URI'. This is a
% globally unique identifier which identifies every file uploaded
% to the service.

% Download the image file you just uploaded by referencing the URI
% img = vispm.download(s); % Return an image

% Now, create a function to negate an image.

f_negate = vispm.function('ifunc;negate');

% The created function is a 'function ticket' which is a reference
% to a function call on the server. Notice that you can click on the
% links displayed to bring up additional information about the function
% on the website.

% Call this function on a newly created image. Notice that a function
% ticket is called just like any other matlab function. The function
% ticket handles all the details of uploading parameters to the server.

% The response is a 'ticket' object. This is a reference to the resulting
% data which is available on the server. Notice that you can click on
% the URI link in the ticket display and see this data on the website.

% Tickets are passed by reference until the underlying data is accessed.
% Try referencing a slice of the data

img = img(16:20,10:200);

% Notice that when the slice is referenced the data is downloaded on
% demand and cached. The next time the data is accessed, the cache
% is quoted.

% If you want to share these results with others, you can share the URI

% Others can read this data and operate further. For example:

f_rotate = vispm.function('ifunc;image.rotate');

% This will rotate the negated image by 25 degrees and display. Notice
% that when the URI was passed, the negated image was not uploaded.
% Rather only the URI was sent since the server already has this data.

% Non-image functions are possible also. Try a hello world function.

% First display help for the function.

vispm.help('ifunc;helloWorld');

% Some functions take longer to run. All of the built-in functions
% called so far operate in blocking mode. All other vispm
% functions operate in non-blocking mode by default. When you call
% a function, you are returned a "transaction ticket". This is your
% receipt for the function call. Let's force a function call to
% return a transaction ticket.

f = vispm.function('ifunc;helloWorld');

% This ticket operates like the others. You can access the underlying
% image just like any other matlab variable. For example, you can
% concatenate:

new_img = [img1, img2];

% Also, you can get notified by email when an operation completes,

f = vispm.function('ifunc;sleep', {'response', 'ticket', 'notify', 'email'});

% You will receive an email when this function completes. You can also
% check the status of a transaction

f.wait();

% Or wait until the transaction is done

% Some functions require credentials on the server. You can get the
% needed credentials by adding 'auth' to the function call. For example:

f = vispm.function('ifunc;image.rotate', {'auth'});

% The following functions require an image as an argument:

vispm.function('ifunc;image.crop', img);

% Demo image files can be downloaded from this demo.

% As you can see, using the.matlab client demo is easy.

% The vispm client demo is available from vispm.org.

% Properties

% vispm URI: http://vispm.org
% licence GPL: <https://www.gnu.org/licenses/gpl.html>
% documentation: vispm.org
% contact: info@vispm.org

% References

from a single image. CVPR 2008, 1, 2


[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and Detterly, Dryad:
Distributed Data-Parallel Programs from Sequential Building Blocks
European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007 2


more than 10,000 image categories tell us? ECCV, 2010. 1, 2

Building Rome in a Day, ICCV, 2009, Kyoto, Japan. 2

[14] Y. Li, D. Crandall, and D. Huttenlocher, Landmark Classification
in Large-Scale Image Collections, ICCV 2009 2

[15] J Shi and C. Tomasi, Good Features to Track, CVPR’94, pp. 593 - 600. 4

[16] P. Felzenszwalb and D. Huttenlocher, Efficient Graph-Based
Image Segmentation International Journal of Computer Vision, Volume 59, Number 2, September 2004 4


