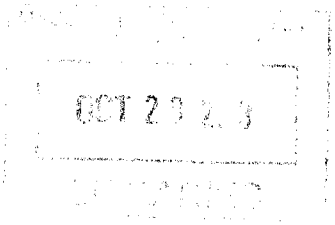
Parallelized Two-Stage Object Detection in Cluttered RGB-D Scenes



by

Sanja Popovic

B.S., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by .....
Leslie Pack Kaelbling
Panasonic Professor of Computer Science and Engineering
Thesis Supervisor

Certified by .....
Tomás Lozano-Pérez
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Parallelized Two-Stage Object Detection in Cluttered RGB-D Scenes

by

Sanja Popovic

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Modern graphics hardware (GPUs) are an amazing computational resource, but only for algorithms with suitable structure. Computer vision algorithms have many characteristics in common with computer graphics algorithms, in particular, they repeat some operations, such as feature computations, at many places in the image. However, there are also more global operations, such as finding nearest neighbors in feature space, that present more of a challenge. In this thesis, we showed how a state-of-the-art object detector, based on RGB-D images, could be parallelized for use on GPUs. By using nVidia's CUDA platform we improved the running times of critical sections up to 38 times.

We also built a two-stage pipeline that improves multiple object detection in cluttered scenes. The first stage aims to achieve high precision, even at the cost of lower recall, by detecting only the less occluded objects. This results in large fraction of the scene being labeled which enables the algorithm in the second stage to focus on the less visible objects that would otherwise be missed. We analyze the performance of our algorithm and lay grounds for the future work and extensions.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Panasonic Professor of Computer Science and Engineering

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my thesis supervisors, Leslie Kaelbling and Tomás Lozano-Pérez, for their unlimited support on the academic as well as personal level in the past four years. I am endlessly thankful to Jared Glover, for being an amazing mentor and collaborator. I would also like to thank Kavya Joshi for late-night company during the process of putting this thesis into writing. Special thanks go to Sabine Schneider for being someone I could lean on and who would always encourage me to keep moving forward. Finally, I would like to thank my family for believing in me and helping me achieve my dreams.

Contents

1 Introduction **9**

1.1 Related Work 9

1.2 Single Cluttered Object Pose Estimation 11

1.2.1 SCOPE Scoring 15

2 Multiple Object Pose Estimation **17**

2.1 Assignment Search 18

2.2 Scoring the Assignment 20

2.3 Two-Stage Approach 22

2.3.1 Evaluation 25

2.3.2 Evaluation analysis 27

3 Parallelization **29**

3.1 CUDA Computing Model 30

3.1.1 Architecture 30

3.1.2 Memory Model 31

3.2 Porting SCOPE to CUDA 32

3.2.1 Scoring Parallelization 33

3.2.2 Round 3 Alignments Parallelization 34

3.2.3 Round 1 Sampling Parallelization 35

3.2.4 Results 37

4 Conclusions and Future Work **38**

List of Figures

1-1 An example of a scene SCOPE is operating on 11

1-2 Observed scene and its corresponding segmentation. 13

1-3 An example of two oriented features found by SCOPE. Red lines represent the direction of the normal. Orange lines represent the direction of the maximal curvature orthogonal to the normal. 14

2-1 A scene on which MOPE typically predicts a better assignment than individual runs of SCOPE. 18

2-2 The eighteen objects in our database. 18

2-3 An example of a hard scene with an ambiguous view of pepper and pear 20

2-4 The two-bowl problem scene along with the top SCOPE pose for the straw bowl 21

2-5 Shape comparison between the blue bowl by itself and blue bowl within the straw bowl. 22

2-6 Comparative view of pear, box and spray bottle 23

2-7 Top: Scene depicting a barely visible object along with the individual object pictures. Bottom: outputs of one-stage MOPE and two-stage MOPE. Different colors represent projected positions of different objects. 24

2-8 Examples of two-stage MOPE outperforming one-stage MOPE in the scenes containing a very occluded object. 25

2-9 The dataset. 26

3-1 2-NN execution diagram. For simplicity, we are only showing 4 threads operating on 16 reference points. 36

List of Tables

2.1 Overview of precision/recall values for different variants of MOPE . . 27

3.1 Comparative times for parallelized sections of SCOPE. All times are in miliseconds per sample/query. 37

List of Algorithms

1 SCOPE Pseudocode 12

Chapter 1

Introduction

In this thesis, we present a system that is capable of detecting objects whose models are known in advance. More specifically, we develop an algorithm that takes as an input a database of 3-D colored object models and an observation in the form of a colored pointcloud and that outputs poses (positions and orientations) of the object instances that are found in the observation. Existing solutions to this problem have already found their place in industry. For example, FANUC robots use vision to perform tasks like bin picking[1]. However, existing systems are constrained to work in environments with few objects, limited clutter, relatively simple objects, etc. These assumptions are not suitable for unstructured environments, for example, a robot that operates in a typical household. Our contribution is a step towards a more general system that is able to handle arbitrary rigid objects even in the presence of clutter. The system is also resilient to noise and ambiguity. In order to achieve usable speed, it takes advantage of modern GPU hardware.

1.1 Related Work

A common approach to recognizing objects in a scene is extracting prominent features on the model and trying to find a matching feature in the observation (a feature is a

[1]Full specification can be found at the following link: http://www.fanucrobotics.com/robot-applications/M-710iC_Bin_Picking.aspx

descriptor of a region - point, segment, etc.).

Two examples of current state-of-the-art methods that use features for general-purpose multiple object detection are given in Tang et al. [17] and Aldoma et al. [1]. The first system uses color histograms, SIFT features [10] and object shape for matching. However, this algorithm relies on segmentation of individual objects which assumes that the objects are somewhat spaced out. This leaves the question of how well this algorithm can perform in very cluttered scenes (the dataset used in the paper had little to no clutter). The work by Aldoma et al. was the main inspiration for our proposed vision system. They rely exclusively on geometric features, primarily on the SHOT feature developed by Tombari et al. [18]. The features used are oriented, so by establishing correspondences between the model and the scene, it is possible to retrieve a potential pose of the object. Unlike Tang et al. who segment the scenes and then label individual pieces, Aldoma et al. find many hypotheses for individual object poses first, and then search for a subset of those hypotheses (containing multiple objects) which fits the scene best in terms of consistency (for example, to avoid overlap between detected objects). We extend the work of Aldoma et al. by breaking down the detection into two stages, one for very visible, easy to recognize objects, and the second one for occluded, not very prominent objects.

The performance of a recognition system is very dependent on the quality of the features the system uses. Two of the more prominent examples of features for range images are Spin Images [8] and FPFH features [14]. Spin Images are point-wise features proposed by Johnson [8]. Each spin image is a 2D array of values computed based on the neighboring points. Intuitively, this image is obtained by spinning a plane around the point normal (thus the name). Fast Point Feature Histograms (FPFH) were developed by Rusu et al. [14]. For each point, one calculates a histogram of the differences between neighboring points' normals to capture the local shape of an object patch. The work has been extended [13] to take into account viewpoints. Muja et al. [12] have combined this work with what they call a binary gradient grid to achieve better recognition performance.

Many computations in computer vision are parallelized via GPUs. Some examples

Figure 1-1: An example of a scene SCOPE is operating on

are GPU implementations of Canny edge detector [11], GPUCV (GPU version of OpenCV library) [2], Viola-Jones face detection algorithm [6] and SIFT [16]. In this work we present an implementation of parallelized hill climbing and K-nearest neighbors search.

1.2 Single Cluttered Object Pose Estimation

As a subsystem for full multiobject detection, our algorithm first searches for one object at a time. Following Glover et al. [5], we call this problem SCOPE - Single Cluttered Object Pose Estimation. The input to SCOPE consists of a 3-D object model, as well as an observed color and depth (RGB-D) image of a scene such as in figure 1-1. The output is a list of fifty 6-DOF pose estimates of the model in the scene. To build a model, we record colored pointclouds of an object from different viewpoints and use different resting positions (we gather 60-90 images per object). We align these scans to get a full colored pointcloud we use as a model. The model additionally contains a set of features: surface normals, FPFH features (computed at every point) and principal curvatures. We also compute range edges; these are points where there is a sudden change in range (3-D depth) from a particular viewpoint (we precalculate range edges for 66 different viewpoints uniformly distributed across the viewsphere). Additionally, the model stores a noise model that predicts range and normal errors for each point. This information encodes the reliability of depth

Algorithm 1 SCOPE Pseudocode

```
1: procedure SCOPE(model, obs)
2:     oversegment the observed RGB-D input
3:     extract features from RGB-D input
4:     /* Round 1 */
5:     assign observed segments to the model
6:     infer model placement from each of the segment matches
7:     cluster pose samples
8:     score samples and prune
9:     /* Round 2 */
10:    score samples (more detailed) and pruning
11:    for i ← 1, num_iter do                    ▷ num_iter is predefined
12:        for i ← 1, n do                       ▷ n = number of samples
13:            match segments to models
14:            perform BPA alignment on segments
15:            if alignment improves score then
16:                accept new scored sample
17:            end if
18:        end for
19:    end for
20:    score samples and prune
21:    /* Round 3 */
22:    cluster samples and prune
23:    perform gradient alignment
24:    assign observation segments to the model
25:    score samples
26:    return scored samples
27: end procedure
```

measurements which SCOPE uses to downweight the information from unreliable regions (for example, reflective surfaces, points near the object boundary, etc.).

Algorithm 1 shows the SCOPE algorithm. It is broken down into three steps or "rounds". It begins by removing the background (e.g. the table supporting the objects) and then it over-segments the observed range image into segments using k-means on point positions, normals and colors. An example of this is shown in figure 1-2. Each segment is a small, uniformly colored region where all the points have almost identical normals. The segmentation is later used in the alignment steps in round 2 and 3. The algorithm proceeds to make thousands of correspondences between the model and the observation based on FPFH features and converts these

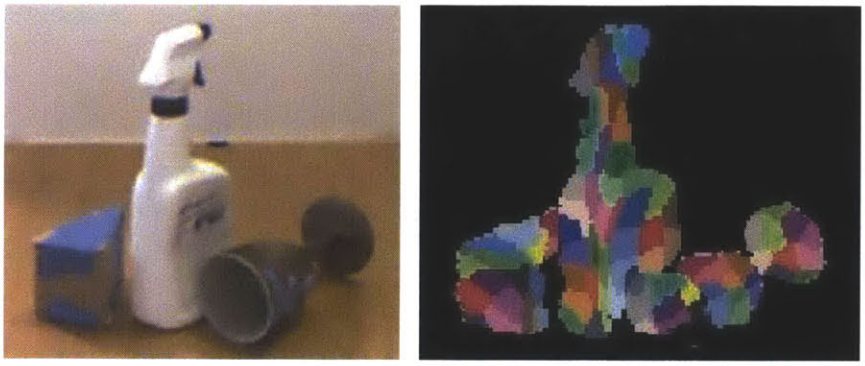


Figure 1-2: Observed scene and its corresponding segmentation.

correspondences into pose hypotheses. Finally, the algorithm performs a three-round elimination of the candidate samples until it finds 50 it considers the best.

To find correspondences in round 1, the algorithm randomly chooses an observed FPFH feature (which can be represented as a 33-dimensional point) and finds its best match on the model using K-nearest neighbors (KNN) where typically $K = 30$. Once the nearest neighbors are found, SCOPE randomly draws a correspondence from the given K features. The sampling distribution is a zero-mean Gaussian based on the distance between corresponding features. The variance σ^2 is computed as the sample variance of FPFH values of all the features in the model database. By establishing a distribution like this, we account for noise, while keeping the property that the closer points (more similar features) are more likely to be sampled. In the previous version of SCOPE, we searched for a fixed number of correspondences regardless of their quality. In this thesis, we reject the correspondence in case the nearest neighbor to the observation point is more than σ away because this is a fairly reliable indicator that the feature we are trying to match is unlikely to belong to a model (or is too noisy to be deemed a good match.) Following Glover et al. [4], SCOPE uses normals and principal curvatures to define a local coordinate frame for each feature (an example of two oriented features is given in figure 1-3). Therefore, once we establish a correspondence, we can generate an object pose sample from it just by aligning a model so that the orientations between corresponding features match.

Currently, round 1 samples up to 2,000 features from which it derives up to 2,000

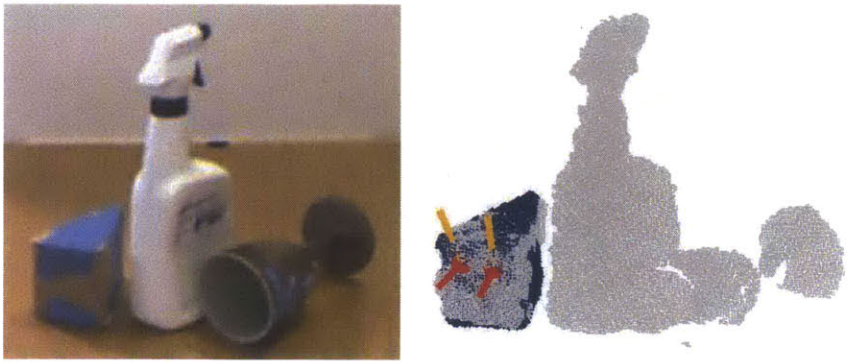


Figure 1-3: An example of two oriented features found by SCOPE. Red lines represent the direction of the normal. Orange lines represent the direction of the maximal curvature orthogonal to the normal.

sample poses. We continue sampling until we either exhaust all the observation points or find 2,000 correspondences. We sample such a high number of poses because SCOPE does not have a prior on the object location in the scene. In other words, there are no assumptions about whether the object is resting on the table, or in which image region it is located.

Because of the number of pose samples, it is extremely likely that many of the poses are almost identical, thus we perform a pose clustering step that iterates over all the sampled poses and groups extremely similar ones (similar samples have a pose difference under 2.5cm and $\pi/16$ radians). Even though it is necessary to sample such a high number of poses, it is practically infeasible to keep all of them because each of the subsequent SCOPE operations has a high cost per sample pose. In order to decide which pose samples to eliminate, SCOPE scores (section 1.2.1) each pose hypothesis and prunes the lowest scoring ones. Currently, round 1 keeps up to 400 (out of 2,000) highest scoring poses after clustering. The actual number might be lower than 400 because the clustering step might eliminate many redundant pose samples.

In round 2, the algorithm performs a more sophisticated scoring of the sample poses. Then, for each of the poses, it finds matches between the observed segments and the model at that pose. This information is used to align the model better the using the Bingham Procrustean Alignment (BPA) method [5]. BPA uses sparse

oriented features to quickly align many samples. The algorithm takes a sample pose, model data and observation data as an input and returns a new pose, which may or may not be better than the original pose (due to noise). All of the new sample poses are then scored again (using the same scoring function as in the beginning of the round) and if the alignments improved the score, the new pose is accepted. This process of aligning and scoring is repeated several times (in practice, the number of iterations is typically 5). Finally, the lowest scoring poses are pruned again.

Due to the alignments, by round 3 many of the poses are very similar so we perform the clustering step and one final round of alignments. We also score and prune so that the number of samples is at most 50. Unlike in round 2, we use hill climbing optimization in round 3. Hill climbing is more expensive than alignment with BPA, because it uses a dense set of point correspondences to directly optimize alignment scores using analytic per-point score gradients. BPA on the other hand uses its own cost function on a sparse feature set and can be thought of as an alignment heuristic. Hill climbing calculates gradients of position and edge scores with respect to local changes in pose. In other words, it is trying to directly optimize xyz, normal and edge scores (explained in section 1.2.1) for each of the samples.

Once the alignment step is done, SCOPE recalculates the matches between observed segments and model points (given their hypothesized pose). The matches might have changed since the beginning of the algorithm if a pose hypothesis changed significantly due to the alignments. This is an addition introduced for the multiple-object algorithm we are proposing and it is unnecessary for SCOPE itself. After this step, the algorithm returns the highest scoring samples along with their scores.

1.2.1 SCOPE Scoring

The scoring function is an extremely important part of the SCOPE algorithm. In round 1, the scoring is much simpler than in rounds 2 and 3 because round 1 operates on a large number of samples for which the more complex scoring would be very inefficient. For each pose, the algorithm randomly chooses 500 projected model points. If more than 20% of those are in the free space (in other words, the predicted depth

of a model point is significantly smaller than the observed depth at that location in the range image), the sample pose is rejected. In round 2, there are 5 different scoring components: xyz score, normal score, visibility score, edge score, and edge visibility score. The final score is a weighted sum of the components. XYZ and normal score measure differences between projected model and observation for ranges and normals for 500 randomly selected model points. The visibility score represents the percentage of predicted model points visible from the camera viewpoint in the given pose. The edge score determines how well the projected model's edges match against the observation edges. Edge visibility, similar to the regular visibility score, is the ratio between unoccluded edge points and the total number of points. It is worth noting that for scoring in these two rounds the 500 points used represent just the fraction (typically $5 - 10\%$) of all the model points. Looking at all the points would be very expensive and many obviously wrong pose samples can easily be ruled out without looking at every model point.

In the final round, the returned samples need to be sorted as well as possible from best to worst fit, so SCOPE performs careful scoring using all the points in the model (usually 5-10 thousand points). We also introduce an additional score component, the segment affinity score. This component measures to what extent the segments assigned to the model respect the predicted object boundaries. The original SCOPE algorithm uses additional 4 components, designed to precisely find hard objects in the scene. However, these components are very computationally expensive and our two-stage approach (section 2.3) does not use them yet. We are currently in the process of integrating additional components into our parallelized system.

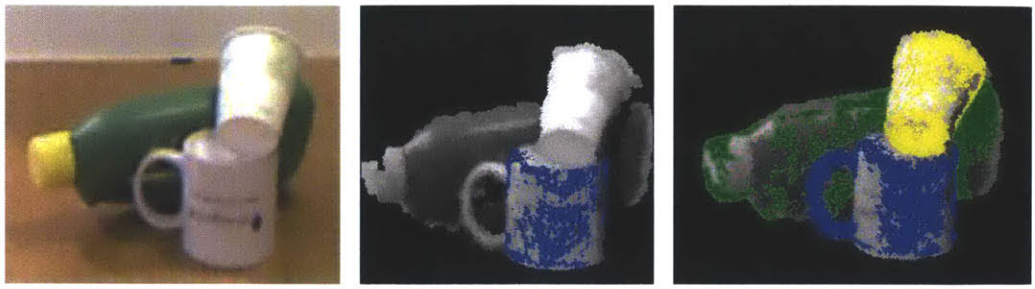
Chapter 2

Multiple Object Pose Estimation

One of the main contributions of this thesis is the Multiple Object Pose Estimation (MOPE) algorithm. MOPE builds on SCOPE and enables finding more than one object in the scene. It assumes we have a database of models of all the known objects in the world. Thus, it only requires an observation scene as an input. The output is a list of model IDs found in the scene along with their poses.

MOPE begins by running a SCOPE pipeline for each object in the database. It saves the top 20 sample poses for each object as ranked by SCOPE. Subsequently, the algorithm determines a subset of the combined set of samples (a set that contains samples for all of the objects) that actually appear in the scene. In further text, we will refer to each hypothesized subset as an *assignment*. In theory, this approach should detect repeated objects given that SCOPE is able to detect both occurrences in its top 20 samples, however we have not yet experimentally confirmed this.

Ideally, the true pose of each object in the scene would be its top ranked SCOPE pose. However, due to noise and a variety of other factors, the correct pose might have a somewhat lower rank. This often happens if there are ambiguities in the scene or an object contains small and relatively noisy parts (like mug handles). For example, when searching for a mug the top-ranked SCOPE pose often assumes an incorrect orientation with the handle at the back (occluded). However, when running MOPE, the detected mug is correctly oriented because this assignment explains the scene better (it covers a larger portion of the scene).



(a) Scene (b) SCOPE output (c) MOPE output

Figure 2-1: A scene on which MOPE typically predicts a better assignment than individual runs of SCOPE.

2.1 Assignment Search

Since the number of possible assignments grows exponentially with the number of sample poses, it is not practically feasible to try every possible assignment; so we use a randomized search. Following Aldoma et al. [1], we chose simulated annealing [9] to search for the optimal assignment. We have also tried tabu search [3], but have not yet observed any difference in average running time compared to simulated annealing.

Figure 2-2: The eighteen objects in our database.

Our database consists of the 18 objects shown in figure 2-2. Given a scene, we run SCOPE on each object and store the top 20 sample poses. During annealing, we keep track of the chosen objects by storing them as an array of pairs (i, j) where i is the index of the object and j is the index of the sample for a chosen object. Initially, our

array is empty. At each step of annealing, we perform one of the following operations (each with a probability of 0.25):

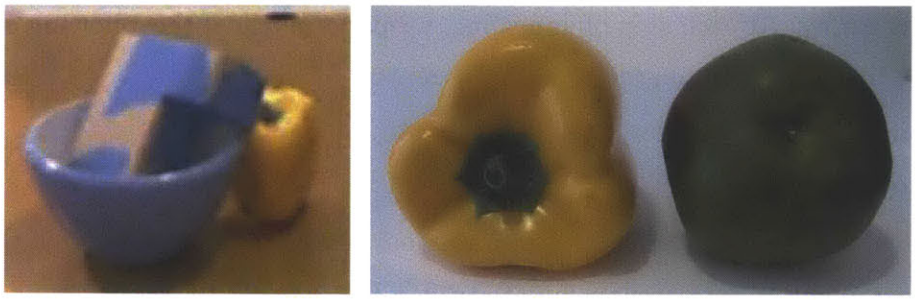
1. Remove an object from the assignment
2. Add an object to the assignment
3. Change the SCOPE sample of a random object (randomly select a pair in the array and select a new value at random for j).
4. Completely switch one of the objects in the assignment (remove one pair from the array and randomly select a new one).

In order to escape local optima, we restart annealing several times. When restarting, we keep track of the best possible assignment, but start from a blank state (no object instances chosen) and restart the cooling schedule on the acceptance threshold.

To ensure that different runs of annealing on each scene yield consistent results, we set the number of iterations to $n = 2,000,000$ and perform $r = 50$ restarts. The cooling schedule we use is $\max(0, A \cdot (1 - i/n)$ where i is the iteration number and $A = 0.5$. In other words, we initially accept the worse assignment with probability 0.5 and the probability gradually drops as the iterations progress.

One important thing to note is the decrease from the 50 samples that SCOPE returns for each object to 20. This is because 50 samples per object created a very large search space and simulated annealing could not consistently find the same assignment for a given scene. We tried doubling both n and r, at which point each scene required on average 5.5 minutes for the annealing part only, but we still could not reach consistency. However, in the vast majority of cases, good SCOPE samples are rarely found beyond position 20 so this approach did not hurt performance from the correctness standpoint.

It is interesting to note that in simple scenes with no ambiguities, even the smaller values of n and r give consistent results. However, there are some scenes that contain objects that MOPE tends to confuse even with larger values of n and r. These objects are usually small and similar in shape (at least their visible portions in the given scene). An example is the scene in figure 2-3(a) containing a yellow pepper. This pepper is geometrically very similar to the pear from the view shown in figure 2-3(b).



(a) Scene (b) Pepper and pear

Figure 2-3: An example of a hard scene with an ambiguous view of pepper and pear

Currently, our system uses colors only for segmentation of the scene so we rely on geometric cues to score sample poses.

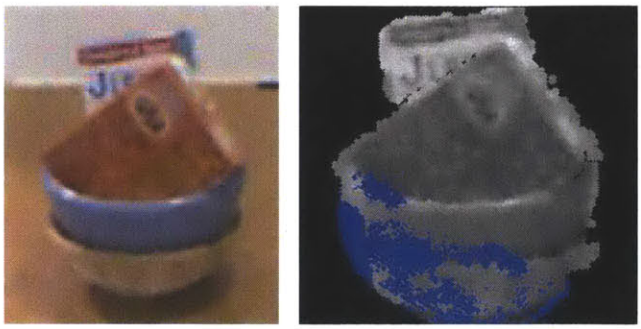
2.2 Scoring the Assignment

We have developed four components to asses the quality of a multi-object assignment.

1. SCOPE fit (S). For each segment in the scene, we find the average SCOPE scores of all the objects covering it and weight by the number of points in the segments (to ensure that larger segments have more influence on the score). The sum across all segments gives the SCOPE score.
2. Explanation penalty (E). This is the percentage of points in the observation that do not belong to any object instance in the assignment.
3. Overlap penalty (O). For each object, we find the percentage of visible model points that overlap with other objects in this MOPE assignment. The average of the penalties per object gives the final penalty.
4. Penalty for the number of objects in the assignment (N).

The final assignment score is calculated as: $w_1 \cdot S - w_2 \cdot E - w_3 \cdot O - w_4 \cdot N$ where w_i's are predetermined weights.

In the first component, SCOPE fit, we have to average SCOPE scores per segment because the borders between objects in clutter are generally fuzzy and it is hard to



(a) Scene (b) Straw bowl labeling

Figure 2-4: The two-bowl problem scene along with the top SCOPE pose for the straw bowl

determine which point belongs to which segment. Additionally, this will somewhat penalize assignments where two different objects are significantly overlapped.

The second component, explanation penalty, will ensure that we try to explain as much as we can in the scene. However, if there are objects in the scene whose models we do not have in our database, this penalty will not force us to erroneously label the objects. Trying to fit a wrong model will highly affect SCOPE fit (unless the model is very similar to the observed object). Additionally, this increases the penalty for the number of objects and those two factors will outweigh the potential benefit from decreasing the explanation penalty.

The overlap penalty helps us avoid assignments that break physical constraints. This component is especially important when objects are hollow and it also balances out the effect of the explanation penalty. For example, in the scene shown in figure 2-4, the depth information is too noisy to clearly show the border between the two bowls. Without the overlap penalty, MOPE's solution to the "two-bowls" problem is typically to first discover the straw bowl (bottom one), but put it in a slanted position (which often happens to be the top ranked SCOPE pose). This covers some fraction of both bowls, but leaves a large amount unexplained. MOPE later proceeds to add the blue bowl to the assignment. The bowl itself has the same height as the blue bowl and straw bowl combination shown in figure 2-5 so geometrically, it makes perfect sense to put the blue bowl in this position. Without an overlap penalty, this

(a) Blue bowl (b) Inside straw bowl

Figure 2-5: Shape comparison between the blue bowl by itself and blue bowl within the straw bowl.

assignment is very desirable since the average SCOPE score does not drop by much (the straw bowl scores reasonably high in that position) and the penalty for number of objects is insignificant. Introducing the overlap score does not solve the "two-bowls" problem, but it at least typically prevents the straw bowl from appearing in an incorrect, physically impossible position. This will also significantly help us in the future, when we increase the importance of color in SCOPE scoring.

We are penalizing the number of objects following Occam's razor where we prefer simpler explanations. This is especially important for large objects whose parts match well with some smaller objects. For example, in figure 2-6 we see a spray bottle whose head is about the same shape as a pear, while the base matches the box. In the scenes where both the head and the base of the bottle are visible, it often happened that MOPE pieced together two objects instead of using one because there was no overlap, and about the same area was covered.

2.3 Two-Stage Approach

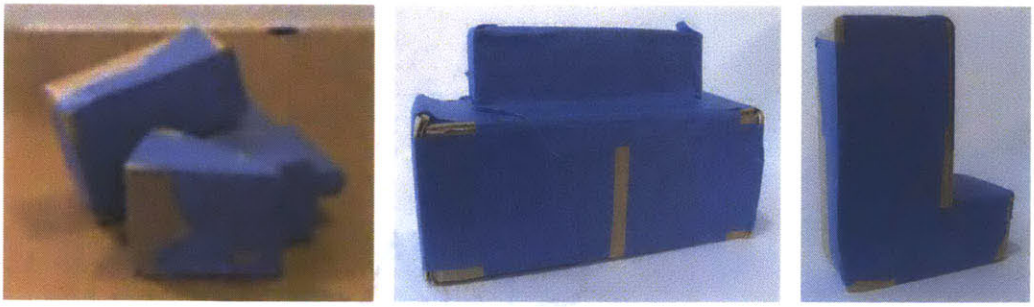
Due to the nature of SCOPE's initial sampling, the MOPE algorithm described above would often miss highly occluded objects, like the rectangular cup in figure 2-7. Since heavily occluded objects cover such a small fraction of the scene, SCOPE's random initial sampling is relatively unlikely to get a guess that is close to the object's actual pose. Additionally, ambiguous objects tend to be mislabeled similar-looking objects

Figure 2-6: Comparative view of pear, box and spray bottle

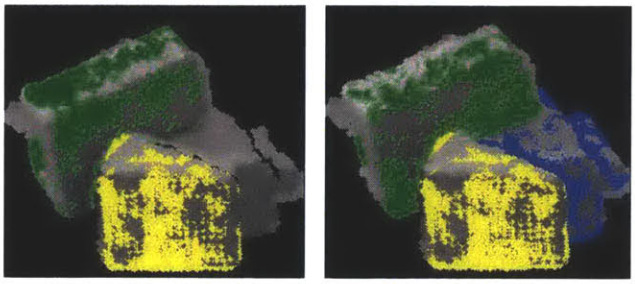
mostly because their SCOPE scores are close. For example, the pepper behind the bowl in figure 2-3 is often labeled as a pear by one-stage MOPE even though SCOPE score for the pepper in the scene is somewhat higher than the one for the pear.

To solve these two problems, we introduce the two-stage approach in MOPE. The first stage is conservative and strives towards high precision, regardless of the recall. This is because we are aiming to label objects that are mostly visible and that comprise a large percentage of the scene. We remove these objects and run the whole pipeline again, but this time trying to achieve a high recall without a significant hit in precision. We omit the second stage if there are not enough unexplained points left after the first stage (the current threshold is 200 points).

Next, we make a decision whether or not we want to accept the result of the second stage. Sometimes the first stage makes a mistake in an object pose, so when the labeled portions of the scene are removed, there is a significant leftover piece of the object in the scene. In that case, stage 2 adds another instance of the same object to cover the leftovers. This is because stage 2 is unaware of the state of the observation and stage 1's results. To fix this, we reassign the segments to the model taking into account the whole scene and reevaluate the MOPE assignment. This will assign the same observation segments to multiple objects, so when scoring, the overlap penalty will be large. After scoring, we accept the stage 2 labels only if the joint score of the entire two-stage assignment is greater than the score achieved by the first stage



(a) Scene (b) Rectangular cup (c) L-Block



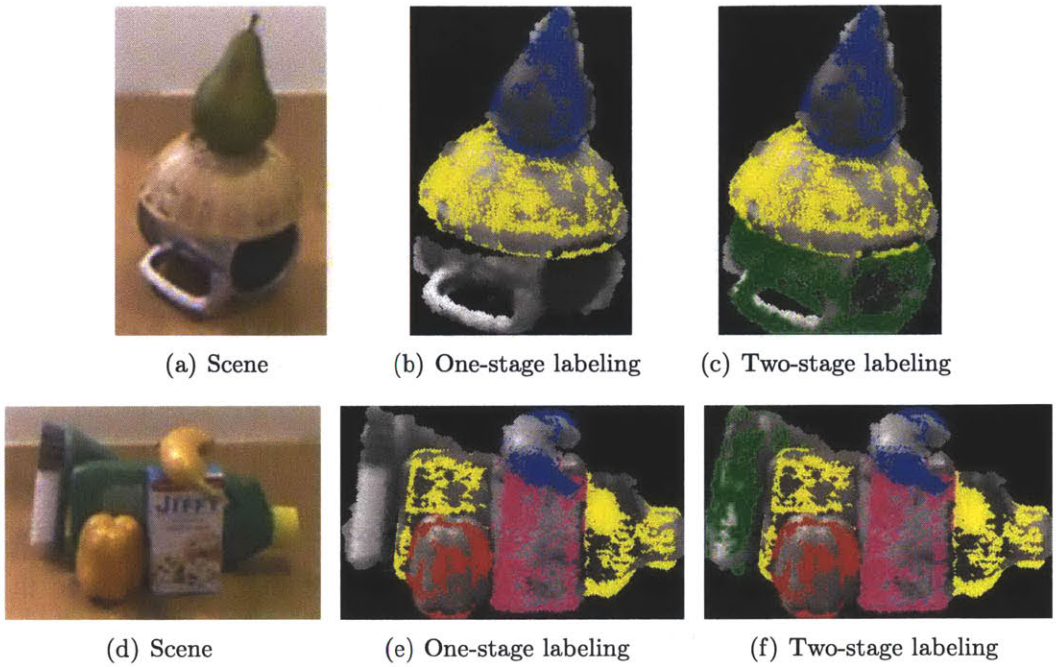
(d) One-stage labeling (e) Two-stage labeling

Figure 2-7: Top: Scene depicting a barely visible object along with the individual object pictures. Bottom: outputs of one-stage MOPE and two-stage MOPE. Different colors represent projected positions of different objects.

assignment only.

The intuition is that after the first stage is finished and points are removed, it is very likely for SCOPE to sample a pose in the region of the actual object in observation and eventually return a good pose hypothesis. For example, in figure 2-7(a), when searching for a rectangular blue cup (the object in the far right, also shown individually in figure 2-7(b)), it is very unlikely SCOPE will ever sample the region where the object is. Additional complication is that object fits well onto the L-shaped block (figure 2-7(c)) in the back of the scene so SCOPE is very likely to report it in the wrong position. However, once we remove the object in the front and the L-shaped block, sampling in the rectangular cup region becomes much more likely. Comparative outputs of one- and two-stage MOPE are given in figures 2-7(d) and 2-7(e). Two additional scenes illustrating the same problem are shown in figure 2-8.

For the pepper-pear problem from figure 2-3, two-stage MOPE first removes the



(a) Scene (b) One-stage labeling (c) Two-stage labeling

(d) Scene (e) One-stage labeling (f) Two-stage labeling

Figure 2-8: Examples of two-stage MOPE outperforming one-stage MOPE in the scenes containing a very occluded object.

objects in the foreground (blue bowl and L-shaped block). In the second stage, the pepper and the pear remain as the only reasonable options and the simulated annealing search is more likely to find the higher scoring MOPE assignment that places the pepper rather than the pear at the back of the bowl.

We achieve different behaviors across the two rounds by using a different set of scoring components for each. Namely, for round 1, we do not use an explanation penalty. This forces MOPE to choose only objects where the gain in SCOPE fit will outweigh the penalty of introducing an additional object. In other words, it makes sure that we do not make errors by choosing mediocre fitting objects.

2.3.1 Evaluation

In order to quantify the performance of our system, we ran both two-stage and one-stage MOPE on all 30 labeled scenes in our dataset (figure 2-9) and measured precision and recall.

Figure 2-9: The dataset.

The baseline algorithm we used for evaluation was one-round MOPE with full set of SCOPE scoring components (10 components compared to the 6 used in this thesis) and only three MOPE components: SCOPE fit, explanation penalty and object number penalty). This is the algorithm we presented in our earlier work [5] and it achieved $p = 83.8\%$ precision and $r = 73.3\%$ recall. In terms of recall, this algorithm is outperforming Aldoma et al. [1] on the same dataset ($p = 82.9\%$, $r = 64.2\%$). Our goal for the two stage algorithm was to either improve on recall, or achieve roughly the same performance, but with a simpler scoring function in SCOPE. For the sake of clarity, we will refer to the SCOPE version used in this thesis as "simple SCOPE" and SCOPE with all of the scoring components as "full SCOPE". Unlike in the original one-stage MOPE from [5], for all the experiments in this section we use the overlap penalty.

To get a better idea of how important the expensive SCOPE components are, we ran one-stage MOPE with simple SCOPE. As expected, the numbers were lower than the baseline ($p = 72.6\%$, $r = 64.2\%$), but high enough to confirm that these components are mainly in place for detecting difficult objects. We then ran only the first stage of two-stage MOPE. We got a reasonably high precision ($p = 79.6\%$) and low recall ($r = 42.5\%$). This is exactly the behavior we expected. By not applying the explanation penalty, we focus only on the objects we are very certain about. This is reflected in high precision, which is comparable to our previous system with full SCOPE. We then ran the complete two stage pipeline and the results we obtained

	Baseline	One-stage	Two-stage, stage 1	Two-stage, complete
Precision	83.8%	72.6%	79.6%	78.0%
Recall	73.3%	64.2%	42.5%	47.5%

Table 2.1: Overview of precision/recall values for different variants of MOPE

were significantly worse than expected ($p = 78.0\%$, $r = 47.5\%$).

2.3.2 Evaluation analysis

We do not yet have a good method for training weights used for scoring and we believe that the lack of proper training is the main cause of the poor performance we measured.

Our goal is to find 6 weights for the SCOPE components and 4 weights for the MOPE component that will yield best results. These weights might also differ between stage 1 and stage 2, for a total of 20 weights. The SCOPE component weights are used to calculate SCOPE fit when scoring MOPE assignments. It is likely that we may want different SCOPE weights for generating SCOPE sampling before computing multi-object scores in MOPE.

The way we currently find weights for round 1 is the following. We assign the initial SCOPE and MOPE weights manually. We then use these weights to find the top 100 MOPE assignments using single-stage MOPE for each scene. Subsequently, we run randomized gradient descent optimizing the weight vector so that we get optimal precision (measured as precision of top-scoring assignments across all scenes) when we rescore the 100 assignments we obtained earlier.

For stage 2 of MOPE, we run the complete stage 1 pipeline and then get SCOPE sample poses for the remaining parts of the scene. We proceed with obtaining the MOPE assignments the same way as in stage 1. For the cost function for stage 2, we use the sum of precision and recall. We then evaluate the performance of MOPE by running the full algorithm on the entire dataset.

The most obvious issue with our method of training is the lack of separate training and testing datasets due to the small size of the dataset. However, we are facing a

greater problem of overfitting to a particular set of MOPE samples. In our dataset, stage 1 typically labels large, easy to detect objects. Collecting MOPE runs on the leftovers of the scenes introduces bias in the weights since stage 2's training dataset might not contain all of the possible objects. Additionally, if we were to not overfit to particular leftovers, we would have to run stage 1 multiple times, presumably with different component weights, which makes generating the training data very expensive.

Another difficulty with training is our dataset itself. In some scenes, stage 1 is able to detect all of the objects so it is not possible to generate stage 2 samples. It is likely that this can be fixed by running stage 1 multiple times with different parameters. Otherwise, we will need to produce a larger dataset, with more heavily cluttered scenes.

The training is particularly difficult because of the number of weights we are trying to optimize for. One of our goals when introducing two-stage MOPE was to reduce the number of weights needed. However, it is possible that we need to reconsider the MOPE components we are using and potentially reintroduce some of the SCOPE components. The four MOPE components used to score the assignments are all very related and somewhat redundant. It is possible that we would be able to obtain better results if we managed to design more decoupled scoring components.

Another potential issue with the system is the algorithm behavior in stage 2. Ideally, we would run stage 2 not only on the highest-ranked sample, but on a few of the top samples to account for potential errors in stage 1. However, this is currently impractical because each stage of MOPE requires running SCOPE on all of the objects.

Lastly, we might need to change the way we perform local search in stage 2 and make the system aware of stage 1 assignments. One potential way to do this is to initialize the annealing with the assignment from stage 1, prevent changes to that part of the assignment and proceed with finding other objects with new SCOPE samples (focused on the unlabeled part of the scene).

Chapter 3

Parallelization

Due to the current nature of MOPE, it is necessary to run SCOPE independently for every object. SCOPE runs are very computationally expensive so in order to make MOPE practically usable, we implemented a series of performance optimizations, out of which the most significant is parallelization. We used nVidia's CUDA platform because it is affordable and offers a large number of cores (the device we tested on has 512 cores) which allows for a very high level of parallelization. The downside of using CUDA is that the way it operates is significantly different from traditional CPUs. In order to get any speedup, we had to significantly change the architecture of SCOPE. The efforts resulted in 3.5-38x of speedup on all of the SCOPE portions we addressed. Overall, the total running time of SCOPE improved by a factor of 6.5.

If we had endless computational power and a good scoring function we would be able to solve the detection problem using brute force, by trying every possible pose for an object (where "every" is an approximation given by a very fine discretization of potential pose values). In practice, such an exhaustive search would require an absurd amount of time, thus we have to revert to random sampling. One might suggest downsampling pointclouds or lowering the number of potential poses (decreasing the granularity of discretization) as a method of shrinking the search space. The first option would not be practical because in the presence of a lot of clutter, some objects are barely visible and downsampling would hide many features and make it impossible to detect such objects. The second option would result in a much lower precision and

even with coarser discretization, there would still remain a large number of candidate poses. Even with our current approach to detection (SCOPE), solving the problem is still very time-consuming so engineering for performance is very important.

3.1 CUDA Computing Model

When we originally implemented SCOPE on CUDA, we noticed a modest speed-up in some functions (around 10x), but to our surprise, there was a noticeable slow-down (around 2x) in other, more complicated functions. This happened because of the different nature of CUDA compared to the traditional CPU parallelism. In order to implement effective CUDA code, it is necessary to understand the architecture well. We will describe the CUDA architecture in this section[1].

3.1.1 Architecture

It is important to notice that GPU cores are significantly weaker than CPU cores. However, unlike CPU threads, it is very inexpensive to launch thousands of GPU threads at the same time. This makes CUDA suitable when a problem can be divided into many relatively small subproblems that can be solved independently. Another significant difference is that the threads are not completely independent. Threads are organized in groups of 32, called warps, that operate essentially in SIMD (Single Instruction, Multiple Destination) fashion. This means that all threads from the same warp have to be on the same line of code. If this is not possible, some threads will be idle. For example, if the code executed by a single branch contains an if-branch and only half of the threads satisfy the condition, those threads will execute the `if` portion, while the other threads are waiting. Only once the first half of the threads has completed their task, the rest of the threads will execute the `else` branch. This means that each *kernel* (all functions executing on the GPU are called kernels) has to be relatively simple, without a lot of branching in order to gain significant

[1]Full documentation can be found at: http://docs.nvidia.com/cuda/index.html

speedup. The issue of threads from the same warp operating on different instructions is commonly referred to as thread divergence.

Threads in general are organized in blocks and blocks are further organized in a grid. When launching a kernel, the user customizes the dimensions of a grid and the blocks it contains. These structures can be one-dimensional, two-dimensional, or three-dimensional which makes CUDA suitable for processing 1D, 2D and 3D data. It is possible to access a thread's x, y and z coordinate within a block, as well as a block's x, y and z coordinate within a grid and their sizes in all three dimensions.

Cores on CUDA are organized into Streaming Multiprocessors (SM) where the number of cores per SM depends on the architecture. When executing a kernel, blocks get assigned to SMs where the number of blocks per SM depends on the number of available SMs. This allows for a simple transition and scalability as more powerful hardware with more cores and SMs is developed.

3.1.2 Memory Model

GPUs under CUDA operate on several memory spaces. The main one is global memory, equivalent of RAM, which is visible to all the threads, regardless of their block. Each block also has a small amount (typically <100KB) of shared memory. Finally, each thread has a very small amount of its own memory. Data in that memory is stored in registers which are a very scarce resource. If the thread runs out of available registers, it will spill over into local memory. Despite the name, local memory actually resides in a special part of global memory. CUDA also has a constant, read-only memory, as well as texture memory. Texture memory resides in global memory, but accessing it is optimized for particular purposes. Usage of all these memory spaces is open to the user and often performance of the program largely depends on the quality of memory usage. Additionally, just like CPUs, GPUs have L1 and L2 cache, but using this cache is left to the GPU itself, rather than the user.

Accessing global memory might take several cycles (if all threads extensively try doing it) which can be a huge performance bottleneck given that each thread executes a relatively short kernel and waiting for a few cycles might be a significant portion

of the thread's life time. Additionally, unless each thread needs to access a separate part of the global memory, it is necessary to use atomic operations which by their nature will impact the performance of the program. In practice, threads typically store intermediate results in their own or shared memory, operate on it and only at the end access the global memory to write the output. However, this requires extra work and careful planning on the user's side.

One more constraint is the lack of dynamic allocation of memory once the kernel is launched. While it is possible to call C's `malloc`, `malloc` is not suitable for the multithreaded environment and repeated calls to it will significantly decrease performance. There is some work done towards creating a suitable dynamic allocator [7], but this is not yet a standard part of CUDA's API.

3.2 Porting SCOPE to CUDA

One of the main bottlenecks of SCOPE is the alignment step in round 3. This function uses many of the score components used for sample scoring which is relatively expensive. Therefore, our first priority was to parallelize the alignments and the scoring function. The most obvious way was to score and align each of the samples operated on in parallel. This was also expected to be the easiest way on the coding side because it would mostly involve launching existing routines on the GPU. Unfortunately, our routines required a lot of dynamic allocations and `malloc` turned out to be even slower in practice than the documentation would imply. As a result, our code was at least an order of magnitude slower than the CPU version.

The next attempt involved preallocating all the necessary dynamic structures before launching the kernels. CUDA allows allocations of 2D and 3D array structures which is what we needed in many places. However, in order to achieve proper memory alignments, CUDA pads the matrices width-wise. On our card, for example, the width has to be a multiple of 512 bytes. Many of our structures were arrays of thousands of points in 3 or 33 dimensions so the rounding would cause a severe memory overhead. Transposing the matrices would result in a huge performance drop because it would

destroy spatial locality of accesses to data. Therefore, we resorted to only allocating one-dimensional arrays, even though operating on those inevitably led to bugs which were hard to track down. However, the performance did improve significantly since the first attempt. Scoring in round 1 became around 10 times faster. Scoring in round 2 was around 4 times as fast and scoring in round 3 became twice as slow compared to the CPU version. In other words, as the rounds progress and their complexity grows, scoring becomes more complex. This showed us the dangers of thread divergence: we only launched one very long kernel that had many different execution paths.

3.2.1 Scoring Parallelization

The third and final attempt needed a complete restructuring of code. The vital observation was that most of the scoring components operate on a per-point basis. In other words, every point contributes a small, independently calculated fraction. Given that all the other data needed for computation is already stored, it is possible to interpret all the samples as $n \times m$ matrix where n is the number of samples and m is the number of points used for scoring. The algorithm then proceeds in stages. First, we launch all the kernels that precompute the necessary data. Then, we have a separate kernel launch for each score component. After this is done, we have several matrices (one for each component) of size $n \times m$, each storing per-point scores for the given component. To obtain per-sample scores, we sum up rows of the matrix. Finally, we sum up individual scores for each sample. This implementation ended up being much cleaner than the previous version because we could dynamically allocate necessary memory right before the kernel needing it was launched. We decided to keep the one-dimensional memory structure introduced in the previous approach.

This approach utilizes the cores available on our device really well. In the beginning rounds, we do not use many points per sample, but we do have many samples. In practice, round 1 has about 500 samples with 2,000 points each which requires 1,000,000 threads. Naturally, not all of these threads will run simultaneously given that we only have 512 cores, but at any given moment, the GPU will be sufficiently occupied. In round 2, there are between 300 and 400 samples each of which also uses

500 points for samplings. This again creates enough work for all the cores. In Round 3, we use only 50 samples, but each sample requires all of the model points for scoring (which is on the order of a few thousands).

The only portion of the algorithm that initially did not keep the cores occupied was summing the individual results. One idea was to launch a thread for each element in each sample and have each thread atomically increment the sum for its sample. However, this would basically serialize summing per sample, so instead we decided to launch only one thread per sample. In all of the rounds, particularly the third one, this was the dominant portion of running time. Therefore, we launched 256 threads per sample, each summing roughly 1/256 of the individual points (in rounds 1 and 2, this was about 2 points per thread, and in round 3 about 40 points). Once all of the threads were done, we used only one additional thread to perform the sum of the remaining 256 numbers (stored in the shared memory). This improved the performance by a factor of 3.

Another important part of the implementation was the block dimension for scoring functions. We opted for $256 \times 1 \times 1$ because all threads from a given block will operate on the same sample. There are two reasons for this. Because all threads within a block run on a single SM at the same time, the algorithm has better cache performance. Neighboring threads in a block access neighboring pieces of memory which makes it possible to load data into cache once and use it several times. The other advantage is using a block's shared memory. If multiple threads reuse the same data from the global memory, we can load it to the shared memory once and avoid the costs of accessing global memory many times.

3.2.2 Round 3 Alignments Parallelization

Round 3 alignments mostly reuse the functionality we developed for scoring functions. The only significant novelty is that gradients per point are arrays of length 7 so the summations performed when calculating the overall gradient per sample from the individual points was somewhat more complicated. Instead of summing individual numbers of a single matrix, for each sample, we had to sum vectors of length 7. We

wanted to perform the same trick we did before, but with 7 times as many threads. The issue with that is the extensive usage of shared memory which is a resource that should not be wasted lightly. Each SM can use only up to 48KB of shared memory at a time. If we were to store $256 \cdot 7 = 1792$ double-precision floating point numbers in shared memory, we would need 14KB of storage space. This means that each SM can process at most 3 samples at the time. Since we perform very careful alignments in this step, lowering precision to single-precision floats was not preferred. Instead, we did not launch additional threads for each of the vector dimensions and just reused 256 threads we had at our disposal. This approach is also preferable because it allows the problem to scale well as the length of vectors grows.

3.2.3 Round 1 Sampling Parallelization

The last remaining significant bottleneck was sampling in round 1. As described in section 1.2, SCOPE establishes correspondences between the features on the model and features in the observation. For that, it needs K-nearest neighbors search which was very expensive to perform. Even though there are a few open-source GPU K-NN implementations, we decided to keep our system free from third-party libraries because of easier maintenance, better code control and ability to tailor the code exactly to our needs.

The property we are exploiting with K-nearest neighbors search is the fact that the number of nearest neighbors we are interested in is rather low (usually 30). For each point in the observation, the algorithm is launching 128 threads, each in charge of 1/128 model points (which in practice is around 80). Those threads then create independent, sorted arrays (stored in the shared memory) of the closest K neighbors they found. Then, the algorithm releases half of the threads and each thread of the remaining half will merge two arrays (its own and one of the threads that was released). This repeats 8 times, until there is only one thread remaining. A simplified diagram is shown in figure 3-1. Using 128 threads ensures that each thread has enough work to do and allows us to work on several query points in parallel. Spawning more threads would mean that the threads in the first iteration will not have enough points

to build their internal arrays. This number of threads also The exact number of query points is discussed below.

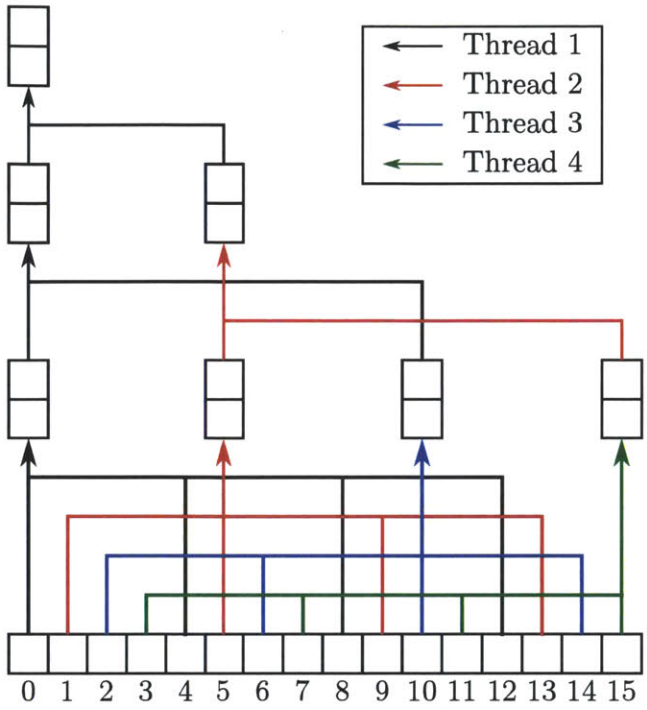


Figure 3-1: 2-NN execution diagram. For simplicity, we are only showing 4 threads operating on 16 reference points.

Parallelizing KNN search resulted only in a factor of 3.5x speedup. This is because the method we used requires a significant amount of shared memory. As we have noted above, each SM can only use up to 48KB at the time. In our case, one thread requires 2 arrays of length K, one for storing 32-bit integers and one for storing single-precision floating-point numbers (also 32 bits). The array of floating-point numbers stores the distances of currently found K-nearest neighbors, while the array of integers stores their indices. For $K = 30$, each thread requires 240 bytes. This means that 128 threads (1 block) require 30,720 bytes total. Additionally, we need to allocate temporary storage for merging two arrays. Because only half of the threads will perform the merge, we need only 50% of additional space, or 15,360 bytes. In total, each block requires 46,080 bytes, which is 45KB. In other words, we cannot run more than 1 block per SM at any given moment. Our GPU has 12 SMs so the result

we obtained make sense given that GPU cores are much slower than CPU cores.

Alternatively, we tried storing some of the structures in global memory, rather than shared memory. However, since each access to global memory is expensive (and the routine requires many of them), the benefit of examining more points in parallel is overpowered by the cost of memory accesses. We also considered releasing pieces of shared memory as threads finish their tasks. However, shared memory has to be allocated statically and can only have a size that is defined as a constant number. The only way to make it non-constant is by specifying the size dynamically during the kernel launch. Unfortunately, this makes it non-constant only from the perspective of host (CPU) function. Once the kernel is launched, the amount of shared memory used is unchangeable.

3.2.4 Results

We ran SCOPE on each of the thirty scenes, three times for each object in the scene and then averaged all of the running times. To account for the variability in number of samples, we recorded the average time per sample (or per query in case of KNN because some queries are rejected) for each run. The results with the speedup factors are given in table 3.1.

		Scoring			
Function	KNN	Round 1	Round 2	Round 3	Alignments
CPU	0.470	0.094	0.463	4.703	111.330
GPU	0.136	0.005	0.019	0.123	7.132
Speed-up	3.46	18.8	24.4	38.24	15.61

Table 3.1: Comparative times for parallelized sections of SCOPE. All times are in miliseconds per sample/query.

Chapter 4

Conclusions and Future Work

In this thesis, we improved the speed of the existing system, SCOPE, as well as introduced a new algorithm, two-stage MOPE, for detecting multiple objects in heavily cluttered scenes. We did many performance optimizations that required significant redesign of preexisting infrastructure. Our main focus was on engineering an end-to-end system and our main goal for the future is to improve upon it and perform more careful training and evaluation.

One of the main problems we encountered when training two-stage MOPE is obtaining a large dataset. Capturing a labeled dataset is a tedious process because it is hard and time-consuming to correctly label the positions of the objects in a pointcloud. However, such a dataset is necessary, especially with the number of scoring components we are considering. One of the potential directions for our future work (in addition to manually collecting more scenes) is developing a synthetic data generator. This approach has been proven to be successful in practice, for example in training the widely-used Microsoft Kinect for skeletal tracking [15]. The generator will somewhat randomly scatter object models in a constrained space to generate a scene (while obeying physical constraints). To account for the camera viewpoint, we will perform ray-tracing from a viewpoint we choose. Being able to generate arbitrarily large datasets would enable us to go one step further and train weights specific to models.

We are also planning on integrating the remaining SCOPE components into our

parallelized system. This will allow our system to have a higher versatility: if necessary, it will be able to perform a slow, but more precise detection. However, if speed is the main concern, two-stage MOPE with the simple SCOPE pipeline (when properly trained) should be able to do the task faster, at the cost of a decreased precision. Additionally, we will further improve the speedups obtained by using CUDA. All of the optimized sections can be improved at least by another factor of 2, by switching from double-precision floating point numbers to single-precision floating point numbers where it is acceptable.

The main future development goal of MOPE is to eliminate the need for running SCOPE on all of the objects in the database. This will lead to tremendous speedups, especially with the growth of the database.

Two-stage MOPE can also be suitable for tracking. If we are certain or almost certain that the observed scene contains certain objects, we can bias the first stage of search towards those objects, and then perform a more careful search in the second stage. For example, if we know that a robot observed a spray bottle and a brush in the scene at a given timepoint, it is not very likely that this changed just a few seconds or minutes later. If the robot moved, it is likely that due to the errors in odometry these objects now have a slightly different pose in the robot's frame of reference. Therefore, in the first stage, we can only search for those particular objects at some approximate locations. Once they are removed from the scene, the second stage can focus on currently unlabeled portions of the observation.

Bibliography

[1] Aitor Aldoma, Federico Tombari, Luigi Di Stefano, and Markus Vincze. A Global Hypotheses Verification Method for 3D Object Recognition. In *Computer Vision–ECCV 2012*, pages 511–524. Springer, 2012.

[2] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. GpuCV: An opensource GPU-accelerated framework for image processing and computer vision. In *Proceedings of the 16th ACM international conference on Multimedia*, MM '08, pages 1089–1092, New York, NY, USA, 2008. ACM.

[3] Fred Glover. Tabu search part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[4] Jared Glover, Gary Bradski, and Radu Bogdan Rusu. Monte carlo pose estimation with quaternion kernels and the bingham distribution. *Robotics: Science and Systems VII*, page 97, 2012.

[5] Jared Glover and Sanja Popovic. Bingham procrustean alignment for object detection in clutter. *arXiv:1304.7399*.

[6] Daniel Hefenbrock, Jason Oberg, Nhat Thanh, Ryan Kastner, and Scott B Baden. Accelerating viola-jones face detection to FPGA-level using GPUs. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 11–18. IEEE, 2010.

[7] Xiaohuang Huang, Christopher I Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. XMalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1134–1139. IEEE, 2010.

[8] Andrew Edie Johnson. *Spin-images: A Representation for 3-D Surface Matching.* PhD thesis, 1997.

[9] Scott Kirkpatrick, D. Gelatt Jr., and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[10] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[11] Yuancheng Luo and Ramani Duraiswami. Canny edge detection on NVIDIA CUDA. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8. IEEE, 2008.

[12] Marius Muja, Radu Bogdan Rusu, Gary Bradski, and David G Lowe. REIN - a fast, robust, scalable REcognition INfrastructure. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2939–2946. IEEE, 2011.

[13] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3D recognition and pose using the viewpoint feature histogram. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2155–2162. IEEE, 2010.

[14] R.B. Rusu, N. Blodow, and M. Beetz. Fast point feature histograms (FPFH) for 3d registration. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3212–3217, 2009.

[15] Jamie Shotton, Toby Sharp, Alex Kipman, Andrew Fitzgibbon, Mark Finocchio, Andrew Blake, Mat Cook, and Richard Moore. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 56(1):116–124, 2013.

[16] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22(1):207–217, 2011.

[17] Jie Tang, Stephen Miller, Arjun Singh, and Pieter Abbeel. A textured object recognition pipeline for color and depth image data. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3467–3474. IEEE, 2012.

[18] Federico Tombari, Samuele Salti, and Luigi Di Stefano. Unique signatures of histograms for local surface description. In *Computer Vision–ECCV 2010*, pages 356–369. Springer, 2010.