ACCELERATION OF MONOCULAR DEPTH EXTRACTION FOR IMAGES

A Thesis in
Computer Science and Engineering
by
Anusha Chandrashekar

© 2014 Anusha Chandrashekar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2014
The thesis of Anusha Chandrashekhar was reviewed and approved* by the following:

Vijaykrishnan Narayanan  
Professor of Computer Science and Engineering  
Thesis Advisor

John Sampson  
Assistant Professor of Computer Science and Engineering

Lee D. Coraor  
Associate Professor of Computer Science and Engineering  
Director of Academic Affairs of Computer Science and Engineering

*Signatures are on file in the Graduate school
Abstract

This thesis evaluates and profiles a monocular depth estimation algorithm in which
depth maps are generated from a single image using a non-parametric depth trans-
fer approach. 3D depth from images has a wide range of applications in surveil-
lance, tracking, robotics and general scene understanding. Recent work shows that
depth can be used as an important cue in visual saliency in order to distinguish
between similar objects. The depth transfer algorithm is evaluated on the Make3D
and NYU datasets and the relative, logarithmic and RMS errors are evaluated for
these datasets. It is shown that the depth transfer algorithm performs better than
the state-of-the-art depth estimation algorithms. A multi-core CPU implementa-
tion of the depth transfer algorithm is profiled in order to determine the compute
intensive stages in the algorithm. A Graphics Processing Unit (GPU) architecture
using NVIDIA Compute Unified Device Architecture (CUDA) for accelerating the
execution time of the bottleneck is proposed. The architecture makes efficient use
of the GPU threads and memory which results in significant speed up. The GPU
implementation is compared with the multi-core CPU implementation and it is
shown that the proposed GPU architecture is capable of accelerating the algorithm
by upto 4.3x (depending on image size) than the CPU-based implementation. A
fast depth estimation technique is proposed to accelerate the computation of depth
of moving objects in a video sequence. This method achieves significant speedup
over the CPU and GPU implementation of the depth transfer algorithm, with pro-
cessing rates that are closer to real time. The depth values from the fast depth
estimator is compared to the ground truth depth values to show that the RMS
error is significantly low and within an acceptable range.
# Table of Contents

List of Figures vii

List of Tables ix

Acknowledgments x

1 Introduction 1
   1.1 Problem Statement ............................................. 3
   1.2 Related Work .................................................. 4
      1.2.1 Monocular Depth Estimation ............................. 4
      1.2.2 GPU acceleration of Belief Propagation .................. 4
   1.3 Research Contributions ....................................... 5
   1.4 Organization of this thesis ................................... 5

2 Depth Extraction from Image using Non-Parametric Sampling 7
   2.1 Depth Transfer Algorithm ...................................... 7
      2.1.1 Candidate Matching ....................................... 8
      2.1.2 Candidate Warping ....................................... 9
      2.1.3 Depth Optimization ..................................... 11
   2.2 Algorithm Performance ....................................... 13
      2.2.1 Advantages ............................................... 13
      2.2.2 Disadvantages ........................................... 14

3 Algorithm Evaluation on Datasets 15
   3.1 Make3D Dataset ................................................ 15
      3.1.1 Evaluation on Make3D dataset ............................ 15
   3.2 NYU Dataset .................................................... 16
      3.2.1 Evaluation on NYU dataset ............................... 16
4 Algorithm Profile and Introduction to GPU
4.1 Algorithm Profile ........................................... 18
4.1.1 GIST .................................................. 18
4.1.2 SIFT Flow ............................................. 19
4.1.3 Depth Inference ........................................ 19
4.2 Introduction to GPU ....................................... 19
4.2.1 CUDA Programming Model ............................ 20
4.2.2 GPU Hardware Model ................................ 21

5 SIFT Flow implementation on GPU .......................... 24
5.1 Dual Layer Loopy Belief Propagation ..................... 25
5.1.1 Range Term .............................................. 25
5.1.2 Data Term .............................................. 25
5.1.3 Message Passing ....................................... 26
5.1.4 Compute Belief ......................................... 29

6 Experimental Setup and Results .............................. 31
6.1 GPU Performance .......................................... 31
6.1.1 GPU execution time .................................... 31
6.1.2 GPU peak bandwidth utilization ....................... 31
6.1.3 Comparison with CPU ................................. 32

7 Fast Depth Estimation for Videos ............................. 36
7.1 Proposed Approach ......................................... 36
7.1.1 Motion Segmentation ................................... 36
7.1.2 Geometrical Derivation ................................. 37
7.1.3 Fast Depth Estimator (FDE) ............................ 39
7.2 Fast Depth Estimator Evaluation .......................... 41
7.2.1 FDE Execution time .................................... 41
7.2.2 Comparison of FDE with CPU based and GPU based implementations .......................... 41
7.2.3 FDE Speedup ............................................ 41
7.2.4 Performance of FDE .................................... 42
7.2.5 Depth maps from FDE ................................. 43

8 Conclusion ................................................... 45
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Depth information in Border Surveillance</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Occluded object tracking using depth information</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Depth information in Saliency</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Depth Transfer Algorithm</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Input image and Top 7 candidates using GIST</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>SIFT Flow algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Candidates warped to input image using SIFT Flow</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Inferred depth for input image from warped candidate depths</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Test images and inferred depth maps for Make3D dataset</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Test images and inferred depth maps for NYU dataset</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Execution-time Profile</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Thread blocks and grid structure in CUDA</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>NVIDIA Fermi Streaming Multiprocessor</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>NVIDIA Fermi Streaming Multiprocessor</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Dual layer loopy belief propagation</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>SIFT Flow Data Term</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Spatial Message Update scheme</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>Dual Message Update scheme</td>
<td>27</td>
</tr>
<tr>
<td>5.5</td>
<td>Sequential Forward Message Update scheme</td>
<td>28</td>
</tr>
<tr>
<td>5.6</td>
<td>Sequential Reverse Message Update scheme</td>
<td>29</td>
</tr>
<tr>
<td>5.7</td>
<td>Diagonal Forward Update scheme</td>
<td>29</td>
</tr>
<tr>
<td>5.8</td>
<td>Diagonal Reverse Update scheme</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph of Image size vs GPU execution time for SIFT flow</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>Graph of GPU peak bandwidth utilization for different image sizes</td>
<td>33</td>
</tr>
<tr>
<td>6.3</td>
<td>Graph of comparison of CPU and GPU performance</td>
<td>34</td>
</tr>
</tbody>
</table>
6.4 Input image, warped candidates from SIFT flow GPU implementation and inferred depth map

7.1 Motion Segmentation mask

7.2 Geometric model for estimating depth of man walking away from camera

7.3 Fast Depth Estimator

7.4 Comparison of original depth, depth transfer depth and FDE depth intensities for 579x430 image size

7.5 Comparison of original depth, depth transfer depth and FDE depth intensities for 135x100 image size

7.6 Original image, ground truth depth map and depth map from FDE for frames t and t+6
List of Tables

3.1 Error metrics for Make3D dataset .......................... 16
3.2 Error metrics for NYU dataset .......................... 17
4.1 Profile Results ................................................. 19
5.1 Pseudo code for message passing .......................... 28
7.1 Depth estimation time per frame for Fast Depth Estimator .......................... 41
7.2 Comparison of depth estimation execution times for different image sizes .......................... 41
7.3 Speedup from FDE for different image sizes .......................... 42
Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor, Dr. Vijaykrishnan Narayanan, for his continuous support and encouragement, patience and effective guidance throughout my research. I would also like to thank my committee member, Dr. John Sampson for his insightful comments and guidance.
I thank my mentor, Dr. Kevin Irick for his constant motivation and inspiration. His guidance has been of immense help and I appreciate all his efforts and contributions towards my research.
I would also like to thank Nandhini Chandramoorthy, Karthik Swaminathan and all my colleagues at the Microsystems Design Laboratory (MDL), at Penn State, who have always been there to help me at every stage of my research. I express my sincere appreciation to the staff of the Department of Computer Sciences and Engineering for their constant support in my administrative matters and the CSE Helpdesk for their prompt service.
I am grateful to family, especially my parents, for their relentless support and encouragement, my sister, who has always been my source of inspiration and my aunt who has always been there for me.
Lastly, I thank all my friends for their constant encouragement and support.
Introduction

Depth perception from an image or sequence of images is a fundamental and extensively researched problem in computer vision. Scene depth has a wide range of applications such as 3D scene reconstruction [1, 2, 3, 4], stereoscopic view synthesis, robot navigation and surveillance. As shown in figure 1.1, depth information can be useful in border surveillance by using a single camera to scan a large area and send an alert if an object is in close proximity of the camera. Depth information can also be used to track occluded objects [5] as shown in figure 1.2. An object which is farther away from the camera and is being occluded by an object nearer to the camera, can be continued to be tracked by using the depth information of the occluded object from previous frames. Depth is also used as a cue in saliency [6] where it is approximated that objects closer to camera are more salient as shown in figure 1.3.

Depth estimation typically consists of active techniques such as using Time of Flight (TOF) camera’s (Swiss Ranger) and structured light techniques (Kinect depth camera) as well as passive techniques using two calibrated cameras for binocular stereo, one camera for monocular and multiview stereo. While active depth estimation techniques are more accurate, have robust estimation in non-textured regions and have higher processing rates (upto 100 fps), they have high power consumption (greater than 10W), high cost and limited measurement range (less than 5m for Kinect). Active techniques also generate low resolution depth maps (upto 640x480), are non-robust in highly textured regions and are sensitive to optical noise, interference and differences in object reflectance. Passive depth estimation techniques, on the other hand, can generate high resolution depth maps, are robust
in highly textured regions and have low cost and low power consumption (less than 2W). However most of these techniques are computationally intensive and can be
non-robust in texture-less regions, repetitive patterns and surface discontinuities. Stereo depth can suffer from occlusion problems as projection takes place from two different view points. Passive depth estimation also has high dependence on scene illumination.

Traditional depth estimation techniques employ two cameras to compute depth maps from left and right stereo images. Such systems have been shown to work well and many such stereo vision algorithms can run in real-time [7], [8], [9]. However many computer vision applications such as object detection and saliency use a single camera and hence it is desirable to compute depth using a single camera, when used in conjunction with other vision applications [6, 10] as it can be easily adapted into existing vision systems. Monocular cameras are also inexpensive as compared to stereo cameras and are more widely available.

Monocular depth estimation techniques can be broadly categorized into two types: The first type uses scene geometry from multiple images taken at different vantage points called multiview stereo. In this case, the moving camera helps determine the camera parameters using structure from motion (SFM) which allows pixel matching in adjacent frames [11]. A non-translating camera with varying focal length allows depth from defocus estimation [12]. The second type computes depth maps by using images similar to the query image on a scene level from a large database of images [13, 14, 15].

Graphics Processing Units (GPUs) have several computation cores which make them well suited for highly parallel computer vision tasks and are widely used for accelerating vision algorithms. The CUDA programming model facilitates easy GPU programming in which individual thread blocks are scheduled concurrently, in order to process in parallel, the independent segments of an algorithm. A Single instruction, multiple thread (SIMT) architecture is used to concurrently schedule individual thread blocks on multiprocessors that are available from the multiple streaming multiprocessors that are present on a GPU.

1.1 Problem Statement

This thesis evaluates a monocular depth estimation algorithm and profiles it in order to determine the bottlenecks. It proposes a Graphics Processing Unit (GPU) architecture to parallelize and accelerate the bottleneck, which is the belief propagation module in the SIFT flow algorithm. A fast depth estimation technique
is also proposed in order to compute the depth of moving objects in a video at processing rates that are close to real time.

1.2 Related Work

In this section, related work on monocular depth extraction and GPU-based hardware acceleration of belief propagation algorithms is discussed.

1.2.1 Monocular Depth Estimation

Early approaches for depth estimation were semi-automatic and involved human operators [16, 17] and hence were time consuming and costly. In this case, there is no assumption of the underlying scene structure and it allows the user to label which regions in the image are closer or farther away from the camera. The more recent automatic methods employ a supervised learning strategy for predicting depth from a single image [2]. In [18], image depth is estimated from predicted semantic labels. Depth can also be estimated using shape from shading [19] where shape is recovered from gradual variation of shading in the image. In [20], high quality depth maps are obtained by using dark channel prior to effectively remove haze from a single input image. Konrad et al. [13, 14] use non-parametric depth sampling to convert monocular images into stereoscopic images.

1.2.2 GPU acceleration of Belief Propagation

Belief propagation has been widely implemented on hardware in many algorithms that compute disparity maps using stereo vision. In [21], a multiscale belief propagation algorithm is implemented on NVIDIA GTX 280 GPU and achieves almost 34x speedup in the computation of disparity map for real world images. In [22], a hardware efficient tile-based belief propagation algorithm is proposed and implemented on a GPU. This method achieves almost 6.59x speedup as compared to the CPU implementation. [23] implements a hierarchical belief propagation algorithm on GPU and achieves almost 45x speedup as compared to the CPU implementation. In [24], a memory efficient belief propagation algorithm is proposed and implemented on a GPU and achieves 22.8x speedup over the CPU implementation.
1.3 Research Contributions

The primary contributions of this thesis are as follows:

- A monocular depth estimation algorithm (depth transfer) was implemented in software and profiled in order to evaluate the most compute intensive stages in the algorithm.

- A novel GPU-based hardware architecture for accelerating the bottleneck (SIFT flow) of the depth transfer algorithm is proposed. The proposed architecture accelerates the dual layer belief propagation optimization step of SIFT flow by parallelizing the algorithm with the help of GPU threads.

- The performance of the hardware implementation is compared with the software (C++) implementation in terms of execution time in order to demonstrate the advantage of using GPUs over multicore CPUs. The experimental evaluation shows that the GPU-based implementation is almost 4.3x faster than the corresponding CPU-based implementation depending on the image size.

- A fast depth estimation technique is proposed for computing the depth of moving objects in a video sequence. This technique significantly accelerates the depth estimation process and can achieve processing rates that are close to real time. The performance of this fast depth estimator is compared with the CPU-based and GPU-based implementations of the depth transfer algorithm.

1.4 Organization of this thesis

The rest of the thesis is organized as follows:
Chapter 2 explains the various stages in the monocular depth estimation algorithm and discusses its advantages and disadvantages. Chapter 3 evaluates the algorithm on different datasets in order to test its performance in different error metrics. Chapter 4 profiles the CPU-based implementation of the algorithm in order to determine the compute intensive tasks. It also emphasizes on the need for hardware acceleration and introduces the GPU for hardware implementation.
Chapter 5 describes the GPU architecture for accelerating the SIFT Flow algorithm, which is the most time consuming step in the depth map generation task. Chapter 6 evaluates and compares the results of the GPU-based implementation with the CPU-based implementation to show the performance gains of the GPU implementation. Chapter 7 proposes a new and fast depth estimation system in order to compute the depth of moving objects in a video and evaluates its performance against the CPU-based and GPU-based implementations of the depth transfer algorithm. Finally, chapter 8 discusses the conclusion and future work.
Chapter 2

Depth Extraction from Image using Non-Parametric Sampling

2.1 Depth Transfer Algorithm

The monocular depth estimation algorithm in [15] is used to compute the depth maps from single images. This algorithm infers depth maps for single images as well as videos by transferring depth from a database of RGBD images (training images) to the input image. The algorithm is non-parametric, which implies that it does not require explicit definition of a parametric model. The training time for the algorithm is defined as the time required to find the similar candidate images from the database and is very less and in the order of a few seconds.

The algorithm can be broken down into three stages for inferring the depth of a single image. In the first stage, candidate images that are semantically similar to the input image are selected from the database of RGBD images. The candidate depths are then warped in order to align them with the input image in the next stage. Finally, the warped candidate depths are interpolated and smoothed using an optimization technique, and the depth map is inferred from the candidate depths. Figure 2.1 shows this depth estimation pipeline from [15]. The algorithm can further be extended to compute temporally consistent depth maps for videos. This is done by incorporating temporal coherence by using optical flow in the optimization step of the depth inference stage. Moving objects are detected using a motion segmentation algorithm in order to ensure that these objects have the same depth as their contact points with the ground surface. The depth of these
moving objects are also incorporated in the formulation of the objective function for the depth inference stage.

The algorithm makes an important assumption that the semantically similar candidate images have approximately similar depth values in regions which appear to be similar to the input image. Since this assumption may not be true in all cases, a global optimization procedure is performed in order to interpolate and smooth the depth maps.

2.1.1 Candidate Matching

In order to find images from the database that are semantically similar to the input image, the GIST [25] algorithm is used. The GIST algorithm estimates the structure or spatial envelope of a scene by modeling a holistic representation of the scene in terms of a set of perceptual properties such as naturalness, openness, roughness etc. The image is first pre-filtered and then filtered again by a filter bank of Gabor filters. The filter responses are then averaged over a 4x4 non-overlapping grid which then gives the GIST descriptor for the image. The size of the GIST feature descriptor (N) is 4*4*number of Gabor filters.

The GIST features are computed for the input image as well as all images in the database. The images that are closest to the input image are then selected by computing the euclidean distance between the input image and each of the training images. The top K (K=7 has been chosen to be an optimum value in this case) candidate images are then chosen by selecting the candidates with minimum euclidean distance. If G1 (G1 = G11, G12, G13.....G1N) is the set of GIST features for
the input image and G2 (G2 = G2_1, G2_2, G2_3, ..., G2_N) is the set of GIST features for the training images, then the matching score to select the top K candidates is computed as:

\[
\text{Matching Score} = \sqrt{(G_{1_1} - G_{2_1})^2 + (G_{1_2} - G_{2_2})^2 + \ldots + (G_{1_N} - G_{2_N})^2}
\]

![Figure 2.2: Input image and Top 7 candidates using GIST](image)

### 2.1.2 Candidate Warping

In order to achieve pixel to pixel correspondence between the input image and the candidate images, the SIFT Flow [26] algorithm is used. This reduces the search space in the depth optimization stage of the algorithm. SIFT Flow is a scene alignment algorithm, where images with similar scene characteristics are aligned. This alignment is achieved by computing warping operators that map pixels from the candidate image’s domain to the input image’s domain.

As shown in figure 2.3, the SIFT Flow algorithm matches SIFT descriptors [27] between pixels in the candidate images and the input image. SIFT descriptors encode local, salient image structures that are invariant to scale, rotation, illumination and viewpoint differences. In order to extract the SIFT descriptors for each pixel, the neighborhood of each pixel (16x16) is divided into a 4x4 cell. The orientations are quantized into 8 bins in each cell, to get a 4*4*8 = 128 dimensional
SIFT descriptor for each pixel. Thus the image is densely sampled to get pixel-wise SIFT features and is called as the SIFT image. The use of SIFT features allows robust matching across different scene or object appearances. The next step is to match the input and candidate SIFT images, using a discontinuity preserving spatial model in order to allow matching of objects located at different parts of the scene. As SIFT descriptors characterize view-invariant and brightness-independent image structures, matching SIFT descriptors allows establishing meaningful correspondences across images with significantly different image content. In order to achieve dense correspondence to match the SIFT descriptors along the flow vectors, an objective function is formulated to estimate the SIFT flow between the SIFT images. Let \( p = (x, y) \) be the grid co-ordinate of images, and \( w(p) = (u(p), v(p)) \) be the flow vector at \( p \). Assume there are \( L \) possible states for \( u(p) \) and \( v(p) \). \( s_1 \) and \( s_2 \) are the two SIFT images to be matched and \( \varepsilon \) contains the spatial four-neighborhood. The objective or energy function for SIFT Flow is defined in [26] as

\[
E(w) = \sum_p \min(||s_1(p) - s_2(p + w(p)||_1, t) + \sum_p \eta(||u(p)||_1 + ||v(p)||_1) + \\
\sum_{(p,q) \in \varepsilon} \min(\alpha|u(p) - u(q)|, d) + \min(\alpha|v(p) - v(q)|, d)
\]

The first term in the objective function is called as the data term which constrains the SIFT descriptors to be matched along with the flow vector \( w(p) \). The second term is a small displacement term that constrains the flow vector to have minimum effect when no other information is available. The last term is the smoothness term.
which constrains the flow vector of adjacent pixels to be similar. The objective function uses truncated L1 norm \((t, d\) are thresholds\) to account for matching outliers and flow discontinuities. A dual layer loopy belief propagation algorithm is used to optimize the objective function. Furthermore, a coarse-to-fine matching scheme is also used to reduce the execution time and memory usage of the matching process. Figure 2.4 shows the candidate images in figure 2.2 that are warped using SIFT Flow.

![Figure 2.4: Candidates warped to input image using SIFT Flow](image)

### 2.1.3 Depth Optimization

The optimization stage infers the depth map for the input image from the warped candidate depth maps by interpolating and smoothing the candidate depths. This process takes spatial regularization into account and estimates per-pixel depth values for the input image. The objective function to be minimized for this optimization step is formulated in [15] as

\[
-\log(P(D|L)) = E(D) = \sum_{i \in \text{pixels}} E_t(D_i) + \alpha E_s(D_i) + \beta E_p(D_i) + \log(Z) \tag{2.1}
\]

\(E_t\) is the data term, \(E_s\) is the smoothness term and \(E_p\) is the database prior. The parameters \(\alpha = 10\) and \(\beta = 0.5\). \(Z\) is a normalization constant of the probability. The data term is defined as

\[
E_t(D_i) = \sum_{j=1}^{k} W_i^{(j)} [\phi(D_i - \psi_j(C_i^{(j)})) + \gamma \phi(\nabla_x D_i - \psi_j(\nabla_x C_i^{(j)})) +
\]

\]}
\[ \phi(\nabla_y D_i - \psi_j (\nabla_y C_i^{(j)})) \]

The data term measures the closeness of inferred depth map \( D \), to each of the candidate depths \( \psi_j (C^{(j)}) \). \( \phi(x) = \sqrt{x^2 + \varepsilon} \) and is an approximation to the L1 error norm with \( \varepsilon = 10^{-4} \). \( \nabla_x \) and \( \nabla_y \) are spatial derivatives and represent the depth gradients. These gradients are weighted by \( \gamma \).

Since some of the candidate depth values will be more reliable than others, the reliability is modeled by a confidence weighting for each pixel in each candidate image. These weights (\( W_i^{(j)} \) is weight of \( i^{th} \) pixel from \( j^{th} \) candidate) are computed by comparing the SIFT descriptors from the SIFT images of the input and candidates.

Confidence weights = \( W_i^{(j)} = (1 + e^{(\|S_i - \psi_j(S_i^{(j)})\|-\mu_s)/\sigma_s})^{-1} \)

where \( S_i \) and \( S_i^{(j)} \) are SIFT features of pixel \( i \) in candidate \( j \). and \( \mu_s = 0.5 \) and \( \sigma_s = 0.01 \).

The spatial smoothness term enforces smoothness which is highly dependent on the image appearance so that the smoothness is not uniformly applied to the inferred depth, but is uniform in regions with similar texture and non-uniform in discontinuities. A per-pixel weighting of the spatial regularization term is applied such that this weight is large where the image gradients are small and small where image gradients are large. The smoothness term is defined as

\[ E_s(D_i) = s_{x,i} \phi(\nabla_x D_i) + s_{y,i} \phi(\nabla_y D_i) \]

The depth gradients along \( x \) and \( y \) (\( \nabla_x D, \nabla_y D \)) are weighted by sigmoidal functions of image gradients along the same direction, \( S_{x,i} = (1 + e^{(\|\nabla_x L_i\|-\mu_L)/\sigma_L})^{-1} \) and \( S_{y,i} = (1 + e^{(\|\nabla_y L_i\|-\mu_L)/\sigma_L})^{-1}. \) \( \mu_L = 0.05 \) and \( \sigma_L = 0.01 \).

The database prior term guides when the depth inference other terms do not influence the pixel values much.

\[ E_p(D_i) = \phi(D_i - \rho_i) \]

The prior \( \rho \) is computed by finding the average of all the depth images in the database.

Equation 2.1 is minimized by using iteratively re-weighted least squares (IRLS)
which is an unconstrained and non-linear solver. IRLS is a fast method and approximates the objective function by a linear function of parameters and solves the system by minimizing the squared residual. This process is repeated until convergence. For quick convergence in a few iterations, the starting estimate is chosen as the median of all candidate depths. Figure 2.5 shows the output depth map of the algorithm.

![Figure 2.5: Inferred depth for input image from warped candidate depths](image)

### 2.2 Algorithm Performance

This section discusses the advantages and disadvantages of the depth transfer algorithm.

#### 2.2.1 Advantages

The algorithm is non-parametric and does not need explicit definition of parametric model or tuning or refining of parameters depending on the test image. It is a fully automatic algorithm and does not require any user interaction for it to function properly. It requires very less training time in the order of few seconds or milliseconds, depending on the size of the database. The algorithm can be easily
extended to compute depth maps of a video sequence and can infer depth of objects in motion.

2.2.2 Disadvantages

The quality of depth maps inferred is dependent to a very large extent on the quality of training images. Hence it is essential to have a good RGBD database that are similar to the test image, in order to infer a reasonable depth map for the test image. The algorithms fails if an indoor image is trained using all outdoor images and vice-versa. The algorithm can sometimes miss computing the depth of very thin an small structures The algorithm cannot infer depth of objects that are not in contact with the ground in case of videos. The algorithms is computationally intensive and it can take more than a minute on an average, to compute the depth map for a single image. Hence it cannot be used in real-time systems.
Algorithm Evaluation on Datasets

In this chapter, the performance of the algorithm is evaluated on some of the popular RGBD datasets like the Make3D Range and NYU datasets by analyzing three common error metrics: the relative error, logarithmic error and the root mean square (RMS) error.

3.1 Make3D Dataset

The Make3D Range dataset consists of all outdoor images with buildings, trees, roads etc. Ground truth depth has been captured using laser. For evaluation of the algorithm on this dataset, 134 test images and 400 training images were used. Figure 3.1 shows the test images and the inferred depth maps using the depth transfer algorithm.

3.1.1 Evaluation on Make3D dataset

Table 3.1 shows the relative, log_{10} and Root Mean Square (RMS) error for the depth transfer algorithm [15] and the Make3D algorithm [2]. The Make3D algorithm uses a set of visual cues such as texture variations, texture gradients and haze to predict the depth of a single still image. Error metrics are averaged over all images in the test set. As seen in table 3.1, the depth transfer algorithm performs better than the Make3D algorithm.
Figure 3.1: Test images and inferred depth maps for Make3D dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Relative Error</th>
<th>$\log_{10}$ Error</th>
<th>RMS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Transfer</td>
<td>0.361</td>
<td>0.148</td>
<td>15.1</td>
</tr>
<tr>
<td>Make3D</td>
<td>0.370</td>
<td>0.187</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: Error metrics for Make3D dataset

### 3.2 NYU Dataset

The NYU dataset consists of all indoor images of home and office. Ground truth depth has been captured using Microsoft Kinect. For evaluation of the algorithm on this dataset, 200 test images and 1249 training images were used. Figure 3.2 shows some test images and the inferred depth maps using the depth transfer algorithm.

#### 3.2.1 Evaluation on NYU dataset

Table 3.2 shows the relative, $\log_{10}$ and Root Mean Square (RMS) error for the depth transfer algorithm [15] and another state-of-the-art algorithm called the depth fusion algorithm [14]. The Depth fusion algorithm also uses a non-parametric sampling approach. However, in this algorithm, the depth maps are computed...
Figure 3.2: Test images and inferred depth maps for NYU dataset using the median of candidate disparity and smoothed using a cross bilateral filter. Error metrics are averaged over all images in the test set. As seen in table 3.1, the depth transfer algorithm performs better than the depth fusion algorithm too.

<table>
<thead>
<tr>
<th>Method</th>
<th>Relative Error</th>
<th>Log$_{10}$ Error</th>
<th>RMS Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Transfer</td>
<td>0.34</td>
<td>0.131</td>
<td>1.2</td>
</tr>
<tr>
<td>Depth Fusion</td>
<td>0.368</td>
<td>0.135</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 3.2: Error metrics for NYU dataset

Thus the depth transfer algorithm has been seen to perform better than the current state-of-the-art monocular depth estimation algorithms on the datasets that were used for evaluation.
Algorithm Profile and Introduction to GPU

This chapter discusses the execution-time profile of the depth transfer algorithm and explains the need for hardware acceleration of the bottleneck in the algorithm. This chapter also gives a brief introduction to the GPU architecture.

4.1 Algorithm Profile

This section discusses the execution-time profiling results of the software implementation of different stages of the depth transfer algorithm on a CPU. Profiling is an important step in determining the compute intensive stages in the algorithm and hence determine potential candidates for hardware acceleration.

4.1.1 GIST

The GIST algorithm, used in the candidate matching stage of the depth transfer algorithm, was implemented using C Sharp and OpenCV 2.4.2 on a 64-bit Intel Xeon 2.67 GHz CPU workstation with 12 GB RAM. The input image is reduced to a 128x128 image and a 512 dimensional GIST vector is computed for this image. The total execution time for the algorithm on CPU is approximately 200 milliseconds.
4.1.2 SIFT Flow

The SIFT flow algorithm, used in the candidate warping stage, was implemented in C++ and executed on a 64-bit Intel Xeon 2.67 GHz CPU workstation with 12 GB RAM. The total execution time, to compute the warping operators for a 320x240 image, is approximately 29 seconds.

4.1.3 Depth Inference

The depth inference stage infers the depth map of the input image from the candidate depths. The algorithm was implemented in Matlab and executed on a 64-bit Intel Xeon 2.67 GHz CPU workstation with 12 GB RAM. The total execution time on CPU for 10 iterations of IRLS non-linear solver for a 320x240 image is approximately 14 seconds.

Table 4.1 shows the execution-time profiling results.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIST</td>
<td>0.2</td>
</tr>
<tr>
<td>SIFT Flow</td>
<td>29</td>
</tr>
<tr>
<td>Depth Inference</td>
<td>14</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>43.2</strong></td>
</tr>
</tbody>
</table>

Table 4.1: Profile Results

As seen in table 4.1 and figure 4.1, SIFT Flow is the most computationally intensive stage and accounts for nearly 66% of the execution time. Hence this algorithm is the bottleneck in the depth estimation process and there is motivation to accelerate this algorithm in hardware. The SIFT Flow algorithm is thus chosen for hardware acceleration on a GPU.

4.2 Introduction to GPU

Graphics Processing Units (GPUs) are commonly used to accelerate many of the computer vision algorithms such as feature extraction, depth perception and structure from motion, all of which are computationally intensive tasks. GPUs are capable of achieving high performance computing because they consist of large
number of simple processing units which can operate on multiple data elements in parallel. The CPU on the other hand has fewer cores which are more efficient and optimized for sequential serial execution. Many of the vision algorithms are inherently parallel which makes the GPU a good choice for acceleration by processing many pixels in parallel on the large number of cores available in GPUs. The GPUs can mask memory access latency by performing arithmetic operations which eliminates the need for data caches. GPUs can be programmed using Compute Unified Device Architecture (CUDA) programming model, which is a general purpose parallel computing platform. The SIFT Flow algorithm was implemented on a NVIDIA Tesla M2070 GPU.

### 4.2.1 CUDA Programming Model

The parallel portions of the program are executed by CUDA kernels. Each of the scheduled N threads will execute the kernel in parallel. Only one kernel can be executed at a time. The threads are grouped to form thread blocks and thread blocks are further grouped to form grids. Threads in a block are executed on a single streaming multiprocessor (SM) and can share data through shared memory and synchronization. There is a limit on the number of threads in a thread block. For Tesla M2070, this limit is 1024. Each thread has a unique ID within a block.
and each block has a unique ID within its grid, which gives each thread in a kernel, a unique ID. Figure 4.1 shows the arrangements of thread blocks and grids as described in [28].

![Figure 4.2: Thread blocks and grid structure in CUDA](image)

### 4.2.2 GPU Hardware Model

The Tesla M2070 is based on NVIDIA’s next-generation CUDA architecture called Fermi. The Fermi GPU architecture consists of many Streaming Multiprocessors (SMs). Each SM consists of 32 CUDA cores and each core can execute one floating point or integer instruction per clock cycle. Each SM also consists of 32K-word register file, thread control logic, 16 Load-Store units, 4 special function units (SFUs) to handle transcendental and special operations such as sine, cos, exponential and reciprocal. Each SM has a set of 32-bit registers that are shared among the warps and 64K local memory that is shared among the thread blocks. Local memory can be split between L1 cache and shared memory. Shared memory is on-chip and hence provides low latency access to data. The GPU memory system also consists of global memory that is implemented in DRAM and is shared among all threads.
Global memory typically has high access latency. In addition, the GPU also consists of read-only constant memory and texture and surface memory. The code is executed on GPU in groups of 32 threads called as warps. Figure 4.2 shows the GPU memory system and Figure 4.3 shows the architecture of a SM.

![NVIDIA Fermi Streaming Multiprocessor](image)

Figure 4.3: NVIDIA Fermi Streaming Multiprocessor

The Tesla M2070 has 14 SM’s, each consisting of 32 CUDA cores, resulting in a total of 448 CUDA cores. Each core executes a sequential thread in a single instruction, multiple thread (SIMT) way i.e. all threads in the same group (warp) execute the same instruction at the same time. When the CPU (Host) invokes a CUDA kernel in a CUDA program, the blocks of the grid are scheduled to the SMs with available execution capacity.
Figure 4.4: NVIDIA Fermi Streaming Multiprocessor
Chapter 5

SIFT Flow implementation on GPU

This chapter describes the GPU implementation of SIFT Flow algorithm using NVIDIA Compute Unified Device Architecture (CUDA). SIFT Flow uses a dual layer loopy belief propagation algorithm to optimize the objective function for matching pixel-wise SIFT descriptors. Belief propagation is a method of performing exact or approximate inference on graphical models such as Markov Random Fields (MRFs) and Bayesian networks. Belief propagation gives the exact inference when there are no loops in the graph. In case of cyclic graphs, loopy belief propagation provides an approximate inference.

Figure 5.1: Dual layer loopy belief propagation
5.1 Dual Layer Loopy Belief Propagation

In dual layer loopy belief propagation, the x and y displacements of pixels are represented by two interacting planes (u, v) as described in the decomposed model in [29]. From section 2.2, the objective function for SIFT flow is defined as

$$E(w) = \sum_p \min(s_1(p) - s_2(p + w(p))_{t,1}) + \sum_p \eta(|u(p)| + |v(p)|) + \sum_{(p,q) \in \varepsilon} \min(\alpha|u(p) - u(q)|, d) + \min(\alpha|v(p) - v(q)|, d)$$

The first term in this objective function is the data term and is encoded by inter-plane interaction as shown in figure 5.1. The second term is the range term. The last term which is the smoothness term is decoupled from $\sqrt{u_x^2 + v_x^2}$ to $|u_x| + |v_x|$, thus separating the horizontal and vertical flow. This reduces the complexity of message passing from $O(L^4)$ to $O(L^2)$ where L is the of set labels which x and y take values from. In case of image matching, the dual layer allows faster propagation of spatial information by more frequent and cheaper intra-plane updates as compared to single layer which is much more memory demanding. Sequential belief propagation (BP-S) algorithm is used for optimization as it has better convergence.

5.1.1 Range Term

The range term for each pixel in plane u is computed as $\eta(|u(p)|)$ and for each pixel in plane v is computed as $\eta(|v(p)|)$. The range term for the u plane is an m-bit vector and an n-bit vector for the v plane. The range term for both planes are computed in parallel on the GPU.

5.1.2 Data Term

The data term is computed by finding the sum of differences of the SIFT descriptors of each pixel over a (m x n) window as shown in figure 5.2. The SIFT images are loaded into global memory and the data term is then computed by finding the difference between the pixel in SIFT image 1 and a (m x n) window of pixels centered around the corresponding pixel in SIFT image 2. The data term is computed
in parallel for all pixels on GPU. The data term for a 640x480 image with a (9x5) window size will require approximately 13MB of memory. The data term is stored in global memory as shared memory is insufficient for such large amount of data.

Figure 5.2: SIFT Flow Data Term

5.1.3 Message Passing

Message passing is used to compute the value of the label at every pixel. In this algorithm, the label is the x and y displacement which is computed at every pixel. The message passing module consists of computing the belief by updating spatial messages as well as updating dual messages for u and v plane. As shown in figure 5.3, the spatial messages are the messages that are sent by each pixel to its neighboring pixels and are computed by summing up the incoming messages from neighboring pixels. In addition, the corresponding dual message from the dual plane and the range term are also added to compute the spatial message of a pixel. Dual message is the message that is sent by a pixel in one plane to the corresponding pixel in the dual plane. The dual message of a pixel is computed by summing up the spatial messages from the neighboring pixels, the data term and the range term as shown in figure 5.4. The message passing module is run for a large number of iterations until the belief propagation algorithm converges. The pseudo-code for the message update scheme is described in table 5.1.
Figure 5.3: Spatial Message Update scheme

Figure 5.4: Dual Message Update scheme

Figure 5.5 shows the original sequential message update scheme that is imple-mented in the CPU version of SIFT flow. The left to right, top down and dual messages are updated sequentially from the first pixel to the last pixel in the forward message update scheme. This is followed by a reverse update scheme in which the right to left, bottom up and dual messages are updated sequentially from last pixel to first pixel as shown in figure 5.6.

In the GPU implementation of SIFT flow, the messages are updated diagonally, in parallel, in order to speedup the update process. Figure 5.7 shows the diagonal forward update process. The left to right and top down messages of the first
\[
N = \text{total number of iterations} \\
P = \text{total number of pixels} \\
\text{for } n = 1,2,3...N \\
\text{for } i = 1,2,3...P \\
\quad \text{updateLeftToRightSpatialMessage}(i, \text{plane1}) \\
\quad \text{updateTopDownSpatialMessage}(i, \text{plane1}) \\
\quad \text{updateDualMessage}(i, \text{plane1}) \\
\quad \text{updateLeftToRightSpatialMessage}(i, \text{plane2}) \\
\quad \text{updateTopDownSpatialMessage}(i, \text{plane2}) \\
\quad \text{updateDualMessage}(i, \text{plane2}) \\
\text{end} \\
\text{for } i = P,P-1,P-2...1 \\
\quad \text{updateRightToLeftSpatialMessage}(i, \text{plane1}) \\
\quad \text{updateBottomUpSpatialMessage}(i, \text{plane1}) \\
\quad \text{updateDualMessage}(i, \text{plane1}) \\
\quad \text{updateRightToLeftSpatialMessage}(i, \text{plane2}) \\
\quad \text{updateBottomUpSpatialMessage}(i, \text{plane2}) \\
\quad \text{updateDualMessage}(i, \text{plane2}) \\
\text{end} \\
\text{end} \\
\]

Table 5.1: Pseudo code for message passing

Figure 5.5: Sequential Forward Message Update scheme

diagonal pixels are computed in parallel, followed by the left to right and top down messages of the second diagonal pixels and so on. Similarly, the right to left
and bottom up messages of the last diagonal pixels are computed in parallel in the reverse update process, followed by the next diagonal pixels, and so on, as shown in figure 5.8. This novel diagonal implementation of the message passing module allows faster parallel computation as opposed to the sequential computation in the original message passing algorithm.

5.1.4 Compute Belief

After N iterations of message passing, the belief is computed at every pixel, by summing up the spatial and dual messages and range term at each pixel and then
finding the minimum value in this sum. This is computed in parallel on the GPU, by launching number of threads equal to the number of pixels.

The horizontal and vertical displacements are then computed from the belief in order to compute the warping operators in the SIFT flow algorithm.
Experimental Setup and Results

The SIFT flow algorithm was implemented in NVIDIA CUDA 5.5 and executed on Tesla M2070 GPU. The specifications of Tesla M2070 are explained in section 4.2. Software simulation used for accuracy comparison was implemented in Matlab.

6.1 GPU Performance

This section discusses the performance of the GPU implementation of SIFT Flow algorithm. Figure 6.4 shows the output depth map computed using the candidates warped using from the GPU implementation of this algorithm.

6.1.1 GPU execution time

The graph in figure 6.1 shows the SIFT flow execution time when executed on the GPU. As seen from the graph, the execution time increases with increase in image size.

6.1.2 GPU peak bandwidth utilization

The graph in figure 6.2 shows the peak bandwidth utilization of the GPU for different image sizes. The bandwidth for Tesla M2070 is 150 GB/s. Peak bandwidth utilization occurs during the data term computation. The peak bandwidth used remains roughly constant irrespective of the image size because of the high degree of parallelism that is exploited in this computation.
6.1.3 Comparison with CPU

The graph in figure 6.3 shows the comparison of CPU and GPU execution time and the speedup from the GPU implementation over the CPU implementation. As seen from graph, the speedup increases as the image size increases, as more and more parallelism of the GPU is exploited. The speedup achieved is almost 4.3x for a 1280 x 960 image.
Figure 6.2: Graph of GPU peak bandwidth utilization for different image sizes
Figure 6.3: Graph of comparison of CPU and GPU performance
Figure 6.4: Input image, warped candidates from SIFT flow GPU implementation and inferred depth map
Fast Depth Estimation for Videos

Many applications such as video surveillance, do not need the depth information of every pixel in every frame as there is not much change in the depth information in consecutive frames. Hence we can accelerate the algorithm by eliminating redundant computations of static background objects. Moving from the depth of images, to the the depth of objects in a video, we can exploit temporal persistence in videos and detect only moving objects in the frame using motion segmentation algorithms. We can then use geometrical estimation techniques to get the depth of moving objects from apparent object size in remaining frames.

Depth of objects in motion in a sequence of images has a number of important applications in video surveillance, object tracking and obstacle detection. In this section, we propose an approach to estimate the depth of moving objects in a video sequence. This approach works by first computing the depth map of the first frame in a video with a moving object using the GPU accelerated monocular depth estimation algorithm. The moving object in consecutive frames of a video is detected by using a motion segmentation algorithm and a geometrical derivation estimates the object depth from the apparent change of the object size in the image plane.

7.1 Proposed Approach

7.1.1 Motion Segmentation

In order to detect the moving objects in a frame, a simple motion segmentation algorithm from [15] is used. The optical flow [30] is computed for the frame, and
the flow is then thresholded to estimate pixels moving relative to the camera. The threshold value is computed as $0.005 \times \sqrt{\text{height} \times \text{width}}$. The output is a binary motion segmentation mask of moving pixels. Figure 7.1 shows a video frame and the corresponding motion segmentation mask.

\[ s/h = f/d \]

\[ s - \Delta s/h = \frac{f}{d + \Delta d} \]

Hence,

\[ s.d = (s - \Delta s)(d + \Delta d) = h.f(\text{constant}) \]
where \( f \) = focal length of camera,
\( h \) = height of object,
\( s \) = apparent object size,
\( s-\Delta s \) = apparent object size in frame \( t+1 \),
\( d \) = object depth in frame \( t \),
\( d+\Delta d \) = object depth in frame \( t+1 \),

Figure 7.2: Geometric model for estimating depth of man walking away from camera

The value of the constant \((s*d)\) for the first frame is computed \((s_{sample} \times d_{sample})\). The value of \(d_{sample}\) is the mode of the depth values of the moving objects detected in the motion segmentation mask. In the depth maps estimated in [15], the depth intensity as a function of distance is computed on a log scale. Hence this depth intensity is converted from log scale to a linear scale. The new depth \(d_{new}\) of the moving object in consecutive frames is then computed from \((s_{sample} \times d_{sample})\) and new object size \((s_{new})\) as

\[
d_{new} = \frac{s_{sample} \times d_{sample}}{s_{new}}
\]

(7.1)

The graphs in figure 7.4 and 7.5 show the comparison between the true depth values and the depth estimated using the geometrical estimation approach for different
image sizes.

### 7.1.3 Fast Depth Estimator (FDE)

Consider a video sequence of \( N \) frames. As shown in figure 7.3, the proposed fast depth estimator (FDE) system consists of \( M \) identical Processing Elements (PE). The value of \( M \) corresponds to the degree of parallelism desired and will depend on the availability of hardware resources. In this system, \( M = 5 \) and the value of \( N \) is 40. The incoming frames from the camera are inputs to the PEs. The first PE processes frames 1,2...K, the second PE processes frame (K+1), (K+2)...(2K+1) and so on, where \( K \) is a small value such that the variation in the depth of moving objects over \( K \) frames is not very significant. In this system, the value of \( K \) is 8.

Each PE consists of GPU accelerated SIFT Flow + Depth inference module to compute the depth of the 1st frame it receives. This is computed in parallel by all PEs. Each PE also consists of a motion segmentation module to detect the moving objects in consecutive frames, a buffer to store the motion segmentation masks and a geometrical estimation module to compute the depth of the moving objects. We then use geometrical techniques to determine the depth of the objects within the motion segmentation mask by using the estimated depth of the 1st frame and the object size in the remaining frames.

\( T_{init} \) is the initial start latency to stream until frame 33, considering an incoming stream at 30fps (each frame streams after approximately 30ms). \( T_{init} \) is 960ms for a 40 frame video. Since the time required to compute the motion segmentation mask (\( T_{motion} \)) for all frames is far less as compared to the time required to find the depth using SIFT flow and the depth inference algorithm (\( T_{depth} \)), \( T_{motion} \) can overlap with \( T_{depth} \) and we buffer the motion segmentation masks until the depth estimation completes. \( T_{geometry} \) is the time required to compute the new depth \( d_{new} \) using equation 7.1.

The time required to compute the depth of moving objects using the fast depth estimator is given by

\[
T_{frame} = \frac{(T_{init} + T_{depth} + T_{geometry})}{\#frames}
\]

The throughput of the fast depth estimator is given by
Figure 7.3: Fast Depth Estimator

\[ \text{Throughput} = \frac{1}{T_{frame}} \]
7.2 Fast Depth Estimator Evaluation

7.2.1 FDE Execution time

Table 7.1 shows the time required to compute the depth of moving objects for different image sizes.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>$T_{frame} = (T_{init} + T_{depth} + T_{geometry}) / # \text{ frames}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>579x430</td>
<td>$(0.96 + 56.36 + 0.06)/40 = 1.4s$</td>
</tr>
<tr>
<td>135x100</td>
<td>$(0.96 + 6.156 + 0.05)/40 = 0.18s$</td>
</tr>
</tbody>
</table>

Table 7.1: Depth estimation time per frame for Fast Depth Estimator

7.2.2 Comparison of FDE with CPU based and GPU based implementations

Table 7.2 shows a comparison of the time required to compute the depth map using the CPU version of the depth transfer algorithm (SIFT Flow + Depth inference), the depth map computed using the GPU accelerated SIFT flow module and the depth map computed using the fast depth estimator. As seen from the table, the fast depth estimator is significantly faster than the other two approaches.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>SIFT Flow (CPU) + SIFT Flow (GPU) + Fast Depth Estimator time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIFT Flow (CPU) + SIFT Flow (GPU) + Fast Depth Estimator time</td>
</tr>
<tr>
<td>579x430</td>
<td>101s 56.4s 1.4s</td>
</tr>
<tr>
<td>135x100</td>
<td>6.9s 6.1s 0.18s</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison of depth estimation execution times for different image sizes

7.2.3 FDE Speedup

Table 7.3 shows the speedup achieved by using the fast depth estimator as compared to the CPU and GPU accelerated versions of the depth transfer algorithm. As seen from the table, we get almost 70x speedup over CPU and 40x speedup over GPU by using FDE for a 579x430 image.
<table>
<thead>
<tr>
<th>Image Size</th>
<th>Speedup over CPU</th>
<th>Speedup over GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>579x430</td>
<td>70.4</td>
<td>39.3</td>
</tr>
<tr>
<td>135x100</td>
<td>38.8</td>
<td>34.6</td>
</tr>
</tbody>
</table>

Table 7.3: Speedup from FDE for different image sizes

### 7.2.4 Performance of FDE

The graph in figure 7.4 shows the comparison of depth values computed for consecutive frames in a 579x430 video sequence using the fast depth estimator with the ground truth depth (original depth) from a kinect depth camera and the depth values computed using the depth transfer algorithm. There are some frames where irregular lighting and shadows affect the accuracy of FDE as seen in the graph. The RMS error in this case is approximately 7.38%.

![Figure 7.4: Comparison of original depth, depth transfer depth and FDE depth intensities for 579x430 image size](image)

The graph in figure 7.5 shows the comparison of depth values computed for consecutive frames in the same video sequence as used in the graph in figure 7.4. The
resolution of the video sequence is decreased to 135x100. The depth values estimated using the fast depth estimator is compared with the ground truth depth (original depth) from a kinect depth camera and the depth estimated using depth transfer algorithm. Again, the accuracy of FDE is off in frames with irregular lighting and shadows. The RMS error in this case increases to 10.05% as expected.

Figure 7.5: Comparison of original depth, depth transfer depth and FDE depth intensities for 135x100 image size

7.2.5 Depth maps from FDE

Figure 7.6 shows different frames in a video sequence, the ground truth depth map from kinect depth camera and the depth maps computed using the fast depth estimator. We assume that the depth of moving objects is uniform in the depth maps from FDE.
Figure 7.6: Original image, ground truth depth map and depth map from FDE for frames $t$ and $t+6$
Chapter 8

Conclusion

Monocular depth estimation is a cheap and portable alternative to complex stereo depth estimation techniques. However, monocular depth estimation is a slow and computationally intensive process. This thesis evaluates and profiles a monocular depth estimation algorithm in which depth maps are generated from a single image using a non-parametric depth transfer approach. The depth transfer algorithm is evaluated on different RGB-depth datasets. The depth transfer algorithm is observed to perform better than the state-of-the-art monocular depth estimation techniques based on the error metrics computed for these datasets. A Graphics Processing Unit (GPU) hardware accelerator using NVIDIA Compute Unified Device Architecture (CUDA) for accelerating the execution time of the bottleneck (SIFT flow) is implemented. The proposed GPU architecture is capable of running upto 4.3x faster, depending on image size, than the CPU-based implementation. Inspite of the hardware acceleration, the depth transfer algorithm does not achieve real time performance. Hence a fast depth estimation technique is proposed to accelerate the computation of depth of moving objects in a video sequence. This method achieves significant speedup over the CPU and GPU implementation of the depth transfer algorithm, with processing rates that are closer to real time and RMS error values that are significantly low and within an acceptable range.

8.1 Future Work

The scope of future work for this thesis is profound. The possible options are listed below
• Further speedup could be achieved in the depth transfer algorithm by accelerating specific stages such as depth inference stage in order to achieve real time processing rates. Customized accelerators such as Field Programmable Gate Array (FPGA) based accelerators can be implemented for further acceleration of the SIFT Flow algorithm. The power consumption of FPGA based accelerators will also be less as compared to GPU based accelerators.

• Alternatives to the sequential belief propagation algorithm such as the hardware efficient belief propagation proposed in [22] can be explored for further speedup.
References


ev,” IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE


plementation using cuda on an nvidia gtx 280,” in Proceedings of the 22Nd
Australasian Joint Conference on Advances in Artificial Intelligence, AI ’09,
Springer-Verlag, 2009.

efficient belief propagation.,” in CVPR, 2009.


in stereo matching on gpu,” in Signal Information Processing Association
Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific, pp. 1–4,
Dec 2012.

[25] A. Oliva and A. Torralba, “Modeling the shape of the scene: A holistic repre-
sentation of the spatial envelope,” International Journal of Computer Vision,

alignment across different scenes,” in Proceedings of the 10th European


[28] “Nvidia cuda c programming guide,”

[29] A. Shekhovtsov, I. Kevtun, and V. Hlav, “Efficient mrf deformation model
for non-rigid image matching,” in In IEEE Transactions on International


