University of Alberta

TOWARDS REAL-TIME ADAPTIVE SUPPORT WEIGHT STEREO ALGORITHMS

by

Yilei Zhang 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Your file Votre référence
ISBN: 978-0-494-47454-9
Our file Notre référence
ISBN: 978-0-494-47454-9

Abstract

This thesis work is divided into three parts. The first part presents a new local binocular stereo algorithm which takes into consideration plane fitting at the per-pixel level. By dropping the fronto-parallel assumption for aggregation window selection, the pre-computed plane orientation for each pixel is used to guide the adaptive weight cost aggregation in the 3D cost volume. The second part uses CUDA programming language to fully harness the prowess of GPUs and achieve near real-time performance for the compute-intensive adaptive support weight cost aggregation method. The last part explores the multi-view camera setup, combines the adaptive support weight idea with parzen-window based photo-consistency metric to get a local occlusion robust stereo algorithm. A simplified real-time GPU version is also implemented. The experimental results for all three parts are very encouraging.

Acknowledgements

I would like to extend my sincere thanks to my supervisors Dr. Yee-Hong Yang and Dr. Minglun Gong for their academic guidance and mental support throughout the last two years. They have been a constant source of inspirations and ideas.

I am very grateful to the committee members, Dr. Nilanjan Ray and Dr. Duncan Elliott, for their important comments.

I would also like to thank my current and former labmates in Computer Graphics Lab at the University of Alberta, in particular Cheng Lei, Jason Selzer, Danielle Sauer, Daniel Neilson, Gopinath Sankar, Omar Rodriguez-Arenas, Xiaozhou Zhou, Xida Chen, Yufeng Shen. Their insight and company has been very valuable.

Financial supports from NSERC, the University of Alberta, and the Memorial University of Newfoundland are gratefully acknowledged.

Finally and most importantly, I wish to thank my parents. Every little achievement I have gained today would not have been possible without their support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Computational stereo is generally defined as the recovery of the three dimensional characteristics of a scene from multiple images taken from different points of view. It has been an intense area of research for decades in the computer vision community.

Several survey papers published along the track of stereo vision research have marked the developments and propelled the progress in computational stereo research. Barnard and Fischler [1] surveyed existing approaches and ongoing stereo projects at that time, which identifies functional components of the computational stereo paradigm and criteria for performance evaluation. Dhond and Aggarawal [9] published their review work in 1989 to sum up major developments within a decade. A collection of new matching methods, several popular theories that adopted in stereo vision, a hierarchical processing model, and the use of trinocular constraints to reduce ambiguity in stereo, are among things that appear in the paper. From that point on, stereo research has matured, and according to [3], "much of the community's focus has turned from general stereo matching into more specific problems". Occlusion and transparency issues in stereo matching, active and dynamic stereo, and real-time stereo implementations are some of the categories that are pursued after the 90s. Scharstein and Szeliski in their paper published in 2002 [34] not only summarizes many well-known matching methods emerged throughout the time but also proposes an in-depth taxonomy of all genres of methods. They also provide a testbed online that researchers can evaluate their methods against others easily.

This unified way of evaluation further intensifies the research in computational stereo vision.

Stereo correspondences can be determined in a number of ways and constrained by a plethora of constraints. But in general, all methods attempt to match pixels from one image with their corresponding pixels in another image. Local stereo methods and global stereo methods, as categorized by Scharstein and Szeliski, use different models to find the correspondence [34]. Local stereo methods use constraints on a small number of pixels surrounding the pixel of interest, hence the name local. On the other hand, methods that seek to meet constraints from a global perspective are referred to as global methods. Local methods, with its limited order of constraints, are generally more efficient, but suffer from locally ambiguous regions like occlusions, low-textured regions, or other kind of image noise. These limitations are widely recognized [34] [3] [47], but there is still progress every year in pushing the limit of local stereo algorithms, either to achieve better quality or to get better speed.

This thesis focuses on both ends of the research interests for local stereo matching. First a new local stereo algorithm is developed that aims for finer reconstruction result and achieves accuracy comparable to several complex global stereo algorithms. Then in the pursuit of speed, the newest tool for harnessing the graphics hardware power – CUDA – is exploited to implement the top-notch local stereo algorithm on GPU for near real time performance. Last but not least, a new occlusion-handling multi-view stereo algorithm is developed, which is based on local constraints and is implemented on GPU to achieve real-time performance at the trade-off of little reconstruction quality loss.

The rest of this thesis is organized as follows. Relevant works and necessary background knowledge in fields of stereo matching, more details on local stereo algorithms and multi-view stereo algorithm, and general purpose computation on GPUs are discussed in Chapter 2. Chapter 3 presents a new local stereo algorithm,

capable of producing high quality disparity maps with subpixel accuracy. A GPU powered real-time implementation of the best local stereo algorithm is discussed in Chapter 4. A novel multi-view stereo algorithm which strives both for quality and speed is introduced in Chapter 5. Finally, Chapter 6 concludes the thesis and discusses possible venues for future improvements.

Chapter 2:
Background and Related Works

The goal of this chapter provides the background materials, which the research performed in this thesis is based on. The most basic questions are described in Section 2.1: 1) What is the stereo matching problem? and 2) What is the computational model adopted for stereo matching in computer vision? A brief summary of popular stereo matching algorithms as well as some commonly used representations and terminologies is also given.

In the next section, a synopsis of existing local stereo algorithms is presented. The main focus of this thesis is to improve both the quality and speed of a local stereo algorithm. Since the adaptive support weight cost aggregation method plays an essential role in this work, a detailed analysis of it is also given. Section 2.3 gives a brief overview of multi-view stereo algorithms.

Finally, GPGPU (General Purpose Graphics Processing Unit) and CUDA (Compute Unified Device Architecture) are presented. The use of GPU brings real-time performance to many local stereo algorithms, and CUDA provides flexibility and ease of use to GPGPU programming.

2.1 Overview of the Stereo Matching Problem

The parallel alignment of the human eyes and their close proximity to each other ensure that the brain receives two similar pictures from two nearby viewpoints at

the same horizontal level. Far away objects will have small relative displacements while nearby objects large displacements between images observed in the left and right eyes. This is depicted in Figure 2.1, where the eyes are represented by a pair of cameras. The observed orange sphere, which is farther away from both cameras than the green sphere, has a smaller relative displacement than the observed green sphere in the two pictures. The displacement of image locations of an object seen by the left and right eyes is referred to as the *binocular disparity*. Our brain can use binocular disparities by looking at both images to decide their relative distances.

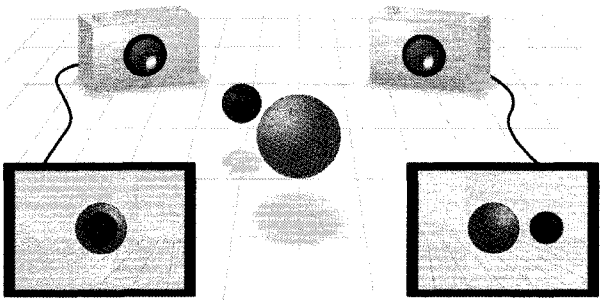


Figure 2.1: The binocular stereo scenario. (Courtesy of Arne Nordmann.)

The human vision system does not give accurate or complete 3D information. Instead, it can decide only the depth information of objects that are visible. As a result, information reconstructed in this way is referred to as $2\frac{1}{2}D$ primal sketch in the pioneering computer vision works by Marr and Nishihara [23]. This $2\frac{1}{2}D$ perception system gives us the sense of depth in the environment we are in, and hence enables us to avoid stumbling over a rock or running into a wall, to grab a cup of coffee knowing how far our hand has to reach out, and to do many other visual tasks we are capable of. When designing a robot, a vision system with similar features is often desired. This is where the stereo matching problem comes into play.

The goal of stereo matching is to match object surface features over two or more images acquired from different viewpoints. After corresponding features are matched, their relative distances or equivalently disparities can then be determined. The stereo matching problem with *two* input images – the counterpart of the human vision system – is called *binocular stereo*. Marr and Poggio in another early vision work [24] suggest three basic steps in computing binocular stereo disparity:

- A particular location on a surface in the scene must be selected from one image;
- The location that corresponds to the same physical point must be identified in the other image;
- The disparity in the two corresponding image points can then be computed.

The search space in step [b] can be greatly reduced by using epipolar geometry described in the following. General position epipolar geometry is illustrated in Figure 2.2. Two pin-hole cameras have their optical centers at O_L and O_R respectively. Each camera has an image plane coloured in light blue. The line connecting O_L and O_R is called the baseline. X is a scene point observed by both cameras at pixel location X_L and X_R respectively. The plane defined by X, O_L and O_R is denoted as the epipolar plane. The intersecting line between the epipolar plane and either image plane is called the epipolar line. Suppose we are searching for the matching point of X_L in the right image. With all of its possible corresponding scene points lining up in the direction of $\overrightarrow{O_L X_L}$ (X_1, X_2, X_3, and the real scene point X, to list a few), their projected locations in the right image plane lie on the epipolar line $e_R X_R$. Thus, it is sufficient to search for the best match along the epipolar line.

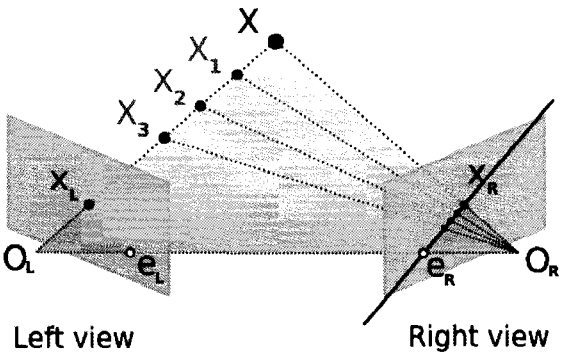


Figure 2.2: Epipolar geometry. (Courtesy of Arne Nordmann.)

The rectified camera setup is simpler than a general position one and is commonly adopted in many stereo vision researches. As depicted in Figure 2.3, two image planes *A* and *B* are arranged so that they are coplanar and collinear on every scanline. By doing that the epipolar lines are parallel to the baseline, and when searching for correspondences the traversal goes along the corresponding scanlines in both images, as depicted by lines *l1* and *l2* in the right illustration of Figure 2.3. Disparity, in this scenario, is defined as the shift of pixels of the corresponding pixels along the same y coordinate in the two input images.

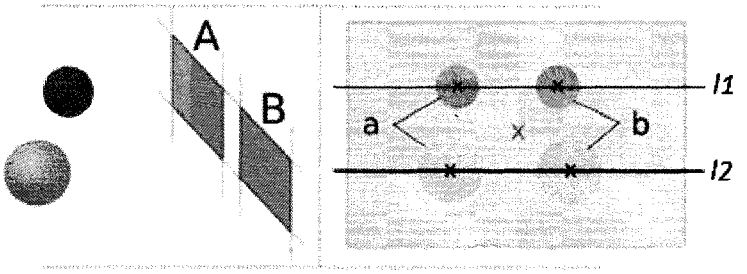


Figure 2.3: Epipolar geometry with rectified cameras. (Courtesy of Arne Nordmann.)

Scharstein and Szeliski use this rectified setup in their popular testbed for dense two-frame stereo correspondence algorithms in the Middlebury Stereo Vision Page [26]. Figure 2.4 is one example dataset adpoted by them. With the

rectified left and right view inputs, for every pixel p_{ref} in the left image (or the reference image), it has a corresponding match on the same scanline in the right image(or the target image). Then at the same pixel location as p_{ref} in the disparity result image, the disparity value between the pair of matched pixels is stored with a proper scaling factor for display purposes. Researchers can evaluate their algorithms by comparing their own results with the provided ground truth and by counting the number of incorrectly labeled pixels.

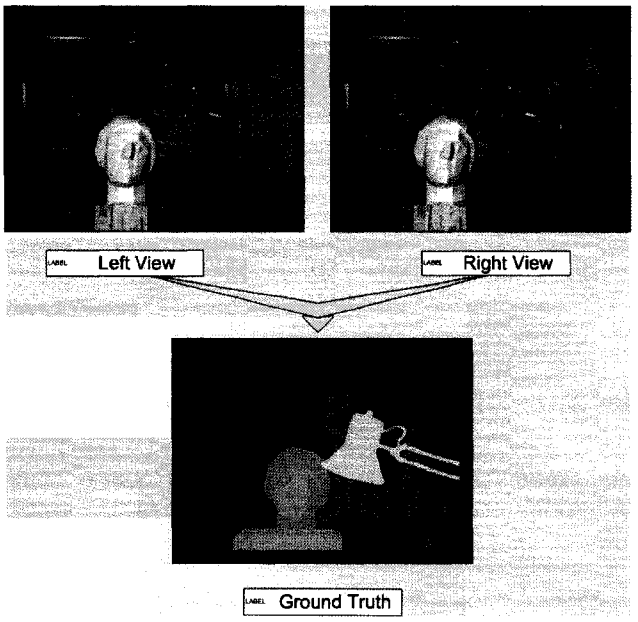


Figure 2.4: The *Tsukuba* dataset used in Middlebury Stereo Vision Page.

The workflow proposed by Marr and Poggio is rather abstract and does not address issues from the computational viewpoint. An up-to-date and detailed computational model for stereo computation has recently been proposed by Scharstein and Szeliski in their taxonomy paper [34]. Based on their taxonomy, most stereo algorithms perform the following four steps or at least a subset of them:

- Matching cost computation. For every pixel p_{ref} from the reference image, a search goes along its epipolar line in the target image and compares the pixel intensities between p_{ref} and p_{target} at each disparity sampling level. The comparison is done with a matching cost function, which decides how well the two pixels match; and a smaller matching cost indicates a better match. The sum of squared differences of intensity values in the red (R), green (G), and blue (B) channels of a pixel pair is one example of matching cost functions. The computed matching cost is then encoded in a 3D *cost volume* $C(x, y, d)$, where (x, y) are the coordinates of the reference image and d is the disparity value. Costs for all the pixels at a certain disparity level d are stored in the same 2D slice and the whole disparity range makes up the third dimension of the cost volume.
- Matching cost (support) aggregation. The quality of single pixel matching cost can be compromised by noise from various sources. So summing up costs over a neighbourhood can help improve the reliability of the matching cost. The set of neighbour pixels with which to compute the matching cost is denoted as the *support*. The simplest support is a square window of a fixed size.
- Disparity computation/optimization. With the generated and refined cost volume, the best correspondences between the reference image pixels and the target image pixels can now be decided. In a local stereo algorithm, the emphasis is on the matching cost computation and the aggregation steps, and a simple winner-take-all disparity selection scheme is used. In contrast, a global approach places more importance in the disparity selection scheme, and many techniques can be used to seek globally optimal disparity choices.
- Disparity refinement. The obtained disparity map then goes through post-processing, such as smoothing, cross-checking, and localizing/refining object edges, to refine the final disparity map.

On a more detailed note, most global optimization based methods minimizes an energy function as suggested in [34]:

$$E(dmap) = E_{data}(dmap) + \lambda E_{smooth}(dmap) \tag{2.1}$$

The data term $E_{data}(dmap)$ measures how well the disparity function $dmap$ agrees with the input image pair. The smoothness term $E_{smooth}(dmap)$ encodes the smoothness of the solution surface. It is usually done by measuring the differences between neighbour pixels' disparities. Once this global energy function is defined, the stereo matching problem is transformed into an energy minimization problem. A variety of algorithms can be used to solve equation (2.1), e.g., the belief propagation [20] [50] [41] based methods yield results of the best quality; the graph-cuts based methods are also studied heavily [21] [46]; the dynamic programming based methods are also very popular [6] [7] [37] [13], but the energy minimization is mostly achieved per scan-line rather than globally over the whole image.

2.2 Review of Local Stereo Matching Algorithms

Global optimization based algorithms has been the dominant approach when the disparity map quality is more concerned. However, the energy minimization framework is computationally intensive and difficult to parallelize, and hence there is room for improvements for local stereo matching algorithms. Indeed, local algorithms are intrinsically parallel. Hence, they can be implemented on current programmable graphics or customized hardware, and thus are widely used in real-time vision applications. This nice feature keeps research on local stereo approaches worthwhile and active.

Out of the four steps described in the previous section, matching cost aggregation is the most crucial part. As for the matching cost computation step, the squared difference is a legitimate choice that performs well under most circumstances, according to Neilson's benchmarking paper [28] on different matching cost functions. Since most of the local stereo algorithms use the same

winner-take-all disparity computation routine, the module that really sets these algorithms apart is in the matching cost aggregation step.

The cost aggregation step basically tries to update every entry in the cost volume based on cost values within its local support regions. How to select the support region varies from approach to approach. The simplest idea is to assume that the neighbourhood will hold the same disparity value with the pixel-of-interest p and the sum over a fixed size square window at each disparity hypothesis d is used to update the cost volume entry $C(x_p, y_p, d)$. This method adopts the smoothness assumption as stated in [24]: disparity varies smoothly almost everywhere, since only a small fraction of the image is composed of boundaries that are discontinuous in depth. However, this aggregation scheme will fail to work at discontinuities. Another setback is that the window size is always hard to decide. As noted by Barnard and Fishler [1], if the window is too small or does not cover enough intensity variation then the disparity estimate does not improve much. On the other hand, if the window is too large, there is a bigger chance for depth discontinuities to present within the window, which violates the smoothness assumption.

2.2.1 Cost Aggregation on Rectangular Windows

Knowing the disadvantages of cost aggregation with square window of a fixed size, researchers have been working on developing more effective adaptive support cost aggregation methods for decades. In an early attempt by Kanade and Okutomi [18], the aggregation window is determined iteratively and has a rectangular shape. Based on the disparity result from the previous pass, the window grows along four directions, i.e. +x, –x, +y, and –y, to minimize the effects of variation of intensity and disparity. Their method is highly dependent on the initial disparity estimation and is computationally expensive.

Geiger et al. [15] stay away from the iterative approach by evaluating two pre-defined windows — one finds the pixel-of-interest on the left border of the window while the other on the right border; the candidate with a better measure is chosen. This idea is later extended by Bobick and Intille [2] and others [5] [14] into the shiftable window method, which considers multiple square windows centered at different locations as depicted in Figure 2.5 and selects the one with the smallest average cost. The pixel coloured black is the pixel of interest, so by varying the position within the aggregation window it can adapt to different boundary situations. However, the proper size of the pre-defined windows still remains a problem.

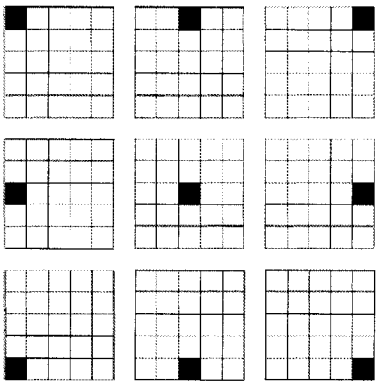


Figure 2.5: Predefined window set in shiftable window method.

The above methods all use rectangular aggregation windows. They are computationally efficient and hence can be incorporated into any existing stereo framework. However, since they do not adapt to local characteristics of the data well, the performance improvement is minimal.

2.2.2 Cost Aggregation with Unconstrained Shapes

To overcome the limitations of rectangular aggregation window, aggregation windows with unconstrained shapes are proposed to better adapt to scene data. One idea is to employ colour segmentation because in many circumstances depth

discontinuity boundaries in a scene also appear as colour discontinuity boundaries [42]. With segmentation information, the aggregation windows can be appropriately selected so that their boundaries will not extend beyond a colour segment. However this approach is not suitable for real-time implementations because colour segmentation itself takes too much time.

A more efficient alternative is to use edge information instead of segmentation information to guide the adaptive window selection [17]. For efficiency consideration, the aggregation process is separated into a horizontal pass and a vertical pass; and in each pass, the locations of colour discontinuities in the horizontal or vertical 1D aggregation window determine how each pixel in the window contributes to cost aggregation.

2.2.3 Cost Aggregation with Adaptive Support Weight

The adaptive support weight (ASW) cost aggregation method addresses the problem of window shape from a different angle. When aggregating the matching cost for a pixel-of-interest p, no clear-cut decision of inclusion or exclusion of a neighbour pixel p' in the support region is required. Instead, a weight is assigned to p' indicating how confident it is to integrate p' during the aggregation process. The concept of using weights makes the support region adapt to the data even though the actual support region is a square window of a fixed size.

The first attempt along this direction is proposed in the paper by Xu *et al.* [49]. In determining the weight for a neighbour pixel within the support region, three cues are used: 1) certainty based on the variance of the error function, 2) colour, and 3) disparity distribution correlation.

The work done by Yoon and Kweon [52] propose a more elegant way to construct the support weight. The support weight for each pixel in the support region is calculated based on the Gestalt Principles, which state that the grouping

of pixels should be based on spatial proximity and chromatic similarity. The original formula proposed is given as follows:

$$w(u,v,m,n) = exp\left(-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta g_{u,v,m,n}}{\gamma_g}\right)\right) \tag{2.2}$$

$$AC(u,v,d) = \frac{\sum_{m,n\in[-r,r]} \begin{pmatrix} w(u,v,m,n)\cdot w'(u,v,m,nri) \\ \cdot C(u+m,v+n,d) \end{pmatrix}}{\sum_{m,n\in[-r,r]} w(u,v,m,n)\cdot w'(u,v,m,n,d)} \tag{2.3}$$

where (u,v) is the pixel of interest; (m,n) is the pixel offset within the local aggregation window; $w(u,v,m,n)$ represents the weight of neighbour pixel $(u+m,v+n)$; d is the disparity hypothesis; $\Delta c_{u,v,m,n}$ is the colour difference between pixel (u,v) and $(u+m,v+n)$; $\Delta q_{u,v,m,n}$ is the Euclidean distance between pixel (u,v) and $(u+m,v+n)$, $C(u,v,d)$ holds the initial matching cost between pixel (u,v) in the left image and $(u-d,v)$ in the right image; γ_c and γ_q are user defined parameters; $AC(u,v,d)$ is the aggregated cost for assigning disparity value hypothesis d to pixel (u,v). The support-weight idea is illustrated in Figure 2.6. The image patch on the left side shows the square support region, and the grayscale image on the right side shows the computed weights with a larger gray level to depict a higher weight. The pixel of interest lies in the lamp arm junction, so the weights suggest that a major part of the lamp arm and lampshade will contribute significantly in the cost aggregation step.

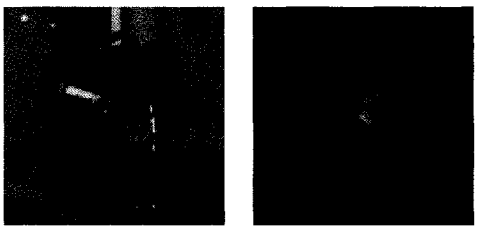


Figure 2.6: An example of weight computation in ASW.

Even with the naïve winner-take-all disparity computation module, ASW gives results comparable to many state-of-the-art global optimization methods, as ranked in the Middlebury stereo vision site [26]. In fact, a majority part of this thesis work is inspired by this approach.

Nevertheless, some drawbacks are still present with ASW. Tombari *et al.* [44] show that the cue of proximity cannot adapt well along depth borders, in low-textured or high-textured regions, regions, or with repetitive patterns. They propose to incorporate the proximity cue with segmentation information so that colour-spatial connectivity can be more efficiently exploited. According to their method, the full support weights are assigned to neighbour pixels lying within the same segment. Their modified weight generation equation is given as:

$$w(u,v,m,n) = \begin{cases} 1.0, & \text{if } Seg(u,v) = Seg(u+m, v+n) \\ exp\left(-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta g_{u,v,m,n}}{\gamma_g}\right)\right), & \text{otherwise} \end{cases} \tag{2.4}$$

Another concern lies in the high computational cost of ASW, as the local support window has to be large enough to effectively encode the neighbourhood information; the suggested window is of size 35×35. Some efforts have been made to accommodate this, *e.g.* in [16] [47], the algorithm is simplified by separating the 2D square cost aggregation calculation into two passes, the first pass along the vertical scan-line and the second along the horizontal scan-line. Their formulation is as follows:

$$T(u,v,d) = \frac{\sum_{m=v,}^{r} w(u,v,m,0) \cdot C(u+m,v,d)}{\sum_{m=m,}^{r} w(u,v,m,0)} \tag{2.6}$$

$$AC(u,v,d) = \frac{\sum_{n=v,}^{r} w(u,v,0,n) \cdot T(u,v+n,d)}{\sum_{n=v+}^{r} w(u,v,0,n)} \tag{2.5}$$

The weights for pixels not on the x and y axes of the aggregation window are approximated using the product of the weights of pixels that are on the axes. This approximation results in a loss of accuracy in the generated cost volume.

2.3 Overview of Multi-view Stereo Matching

As it is generally recognized, using more cameras in stereo can substantially improve the quality of reconstructed depth information because more information is available. For example, multi-view stereo algorithms can deal with the noise problem better than traditional two-camera stereo algorithms and are more robust against occlusion. As summarized by Seitz *et al.* [35] in their survey, given the dense sampling of a scene, the best current multi-view methods use non-linear energy minimization along with visibility handling and silhouette constraints to reconstruct the 3D model.

A major problem that multi-view stereo algorithms face is on how to handle visibilities using multiple images. Due to occlusion, for a certain pixel in the reference view, not all the cameras can see the corresponding physical point in the scene. Therefore, to obtain the optimal disparity map, the matching cost calculation should exclude the cameras that cannot see the corresponding physical point. Seitz et al. [35] classify multi-view stereo algorithms into two categories with respect to visibility handling: geometry-based and outlier-based.

Geometry-based methods need an approximate geometry of the scene or some special assumptions of camera setup to estimate the visibility of a pixel. The voxel colouring method [36] assumes the convex-hull camera setup which means that the occlusion ordering of points in the scene is the same for all the cameras. By evaluating scene points in a near-to-far manner, farther scene points are rejected when they are occluded by validated nearer scene points. Therefore the visibility problem is automatically addressed. This approach leads to a number of plane-sweeping algorithms. For example, techniques by Drouin [10] [11] use an

iterative approach to compute the approximate geometry of the scene and use it to guide the visibility handling in subsequent iterations. Approaches presented by Kutulakos and Seitz [22] along with the works by Sinha and Pollefeys [40] use approximate geometric reasoning (such as visual hull) to infer visibility relationships. This category of algorithms may not work well in a sparsely sampled scene as it is difficult to find a good initial estimate, so the benefits of geometry information may not be realized.

The outlier-based approaches do not require information on scene geometry explicitly. For a particular pixel, only cameras with a better chance of seeing its corresponding physical point are chosen and others rejected. Another nice feature of these techniques is that image noise or the presence of highlight can be treated in a similar fashion as outliers, which are rejected before merging. An early multi-view stereo system proposed by Nakamura *et al.* [27] uses some pre-defined visibility masks in their camera array configuration. For a pixel with a given disparity hypothesis, different predefined visibility masks are evaluated and the best mask is selected for that pixel-disparity combination. Kang *et al.* [19] choose the left or the right half reference cameras in their linear camera array setup; another variant they propose is to use the best half of all reference cameras based on the matching scores. These heuristic outlier-based approaches generally do not give consistently convincing results.

Vogiatzis *et al.* [46] recently present a multi-view algorithm that does not require explicit visibility handling. Instead, an occlusion robust photo-consistency metric is adopted. Photo consistency checking refers to the process of comparing pixels in one image to pixels in other images to see how well they correlate. The variance of the projected pixels from the reference image into the target images is indicative of how well this projection reflects the real depth of the scene point. For any optic ray $\vec{r}$ that goes through certain camera's optic center and intersects with a 3D scene point x, the photo-consistency scores can be computed along $\vec{r}$ with all the other cameras. Then the searching for x along $\vec{r}$ is regarded as a

process of robust model fitting to data containing outliers, which can be caused by occlusion, noise, lack of textures or specular highlights. Their encouraging experimental results suggest the effectiveness of their methods and motivate us to follow their direction and develop a new sparse multi-view cost volume merging approach as described in Chapter 5.

2.4 Hardware Acceleration

GPGPU stands for General Purpose Graphics Processing Unit. While GPUs were originally introduced to unload rasterization-based rendering computations from the CPUs, the graphics hardware industry has made leaps and bounds to overachieve this goal. The most recent GPU chips bear over 400-Gflops computational power, and can be programmed to run SIMD (single instruction multiple data) parallel computations. Therefore they are perfect platforms to implement data parallel applications.

2.4.1 GPU Architecture and GPGPU

The older generations of GPUs were not programmable but hard-wired graphics pipelines. Typically, triangle vertices are transformed, lit, and rasterized into pixels. Then each pixel is shaded with specified lighting and effects, e.g., diffuse lighting, specular exponentiation, fog blending, and frame-buffer blending. Besides defining the scene data input and tweaking with API input parameters, programmers did not have much control over the rendering process. As shown in Figure 2.7, the pipeline is broken down into the following stages:

- Application: This stage provides high-level control to the CPU. It is responsible for passing down the 3D geometry primitives in the form of vertex coordinates, marshalling textures, as well as other organizational works.

- Vertex Transformation: This stage does the vertex position transformation, lighting computations per vertex, along with generation and transformation of texture coordinates.
- Rasterization: The transformed 3D primitives are rasterized into fragments and mapped to the image plane pixels here, with proper depth information computed. The interpolated values for each fragment sent down from the vertex transformation stage also have to be computed.
- Fragment Processing: The final colour for each fragment or pixel is decided in this stage. Besides the interpolated values computed in the rasterization stage, texel calculation, and all other effects that contribute to the final pixel colour is applied here.
- Output: All the fragments sent down the pipeline go through depth test, alpha test, and a series of other tests to decide the final pixel information for the display.

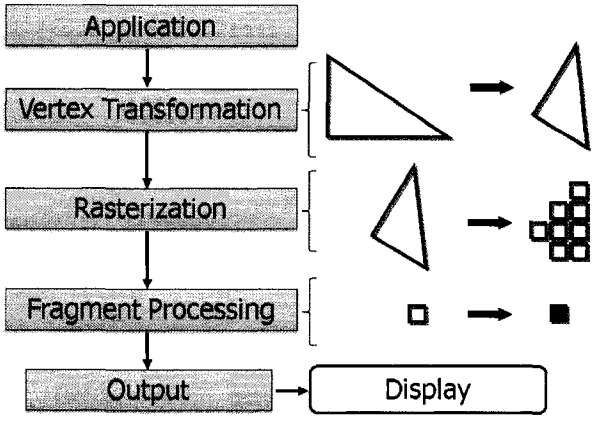


Figure 2.7: A simplified model of a programmable graphics pipeline.

The introduction of programmable graphics pipeline opens a new era of computation not only in computer graphics but also in other areas as well, in particular, computer vision. Each of the operations originally performed in a hard-wired pipeline is now abstracted by its component memory access and mathematical operations. Programmers have the freedom to explicitly define the

functionality of vertex transformation and fragment processing engines based on their needs. The redefined engines can either perform exactly how the hardwired pipeline does or do much more by using customized shader programs, which are programs for controlling the GPUs. A simple example is to include the per-pixel lighting in fragment processing using a fragment shader program.

To adopt this architecture in a general purpose computation, a GPU is often regarded as a stream processor [4]. In the stream processing model, a stream is a collection of records requiring similar computation while kernels are processing functions applied to each element in the stream. A stream processor executes a kernel over all the elements of the input stream, placing the results into an output stream. Hence the mapping from the GPU resource onto the stream processing model is intuitive: encode input streams into textures, use fragment program to execute kernel computations, and store the output stream in an output frame buffer. Figure 2.8 shows how it is done in practice. With the high level graphics API, the GPGPU application defines a screen sized quad as the only primitive to be 'drawn'; the data marshalling stage is where all input data are assembled into textures; then data distribution is automatically achieved by the rasterization hardware; the execution of kernels written in fragment programs follows; finally the result is written back into the frame buffer.

GPGPU has been extensively exploited by many compute-intensive and data parallel projects, such as image processing [12], video encoding and decoding [39], as well as stereo matching [13] [16] [47] [48]. If a GPU – the parallel processing powerhouse – is fully harnessed, one can see tens or even hundreds of times of speed-up over the corresponding CPU counterparts.

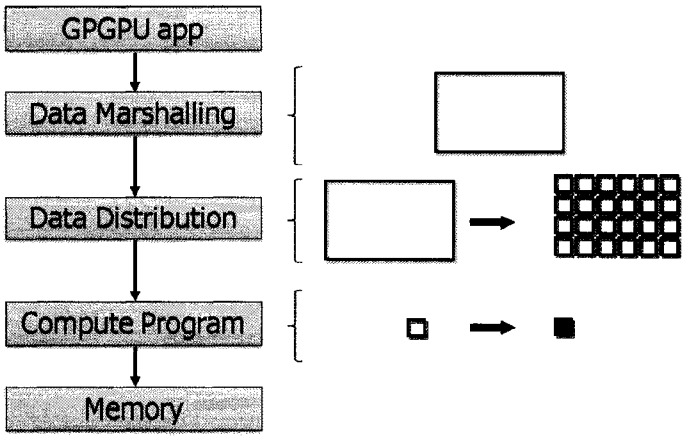


Figure 2.8: Stream processing model with GPU architecture.

2.4.2 CUDA

GPGPU programming with graphics APIs like OpenGL and DirectX can simplify the use of GPU, but there are still limitations as noted in [30]. First, graphics APIs impose a high-learning curve for non-graphics users and also incur overhead when the application is wrapped with graphics API calls. Secondly, the device memory access pattern is highly constrained by a limited number of frame buffers available to the fragment program. Finally, some applications are constrained by the available device memory bandwidth.

Researchers have made many attempts to address these problems, including Brook [4] and Sh [25]. CUDA, the Compute Unified Device Architecture, was introduced by NVIDIA [31] as another alternative and provides unprecedented ease and flexibility in GPGPU programming.

CUDA programming uses a set of C-alike APIs, which means that graphics related knowledge is no longer required to write GPGPU programs. For example, previously the data marshalling is done by encoding the data into textures, therefore appropriate texture related APIs have to be called to initialize a copy of

data. With CUDA, it is as easy as calling *malloc* and *memcpy* for allocating memory and copying the data array.

The thread-batching model adopted in CUDA has a better abstraction of the GPU compute architecture. The batch of threads that execute a kernel is organized as a grid of blocks, as illustrated in Figure 2.9. A *grid* includes all the kernels on the GPU for running a CUDA program. A *thread block* (or simply a *block*), coloured in yellow, consists of a number of kernels that run concurrently on one GPU multiprocessor and can cooperate more tightly by efficiently sharing data through some fast shared memory and by synchronizing their execution to coordinate memory accesses. This (grid ~ block ~ thread) architecture is the direct mapping of (GPU ~ multi-processor ~ stream-processor) structure in the GPU hardware.

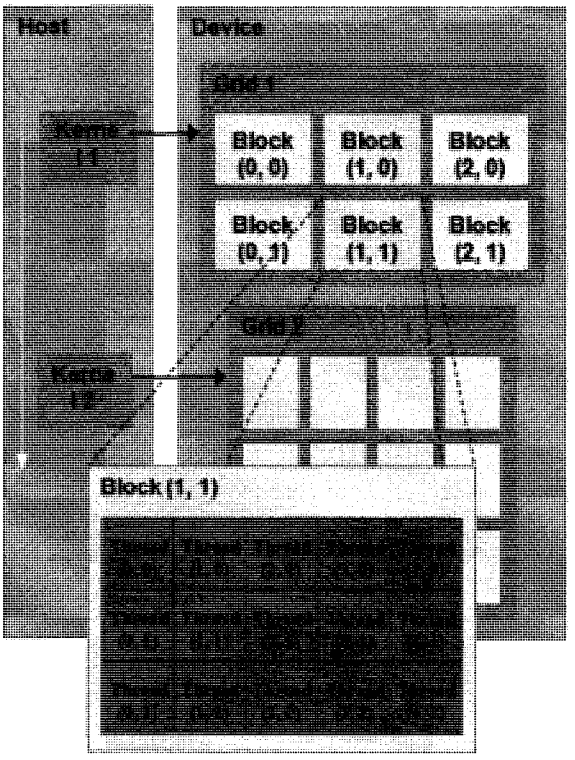


Figure 2.9: Thread-batching model of CUDA.

The memory programming model of CUDA, if carefully adopted, enables more versatile memory access pattern and improves memory access efficiency. Six types of memories are exposed and each serves its own purpose, as shown in Figure 2.10. *Shared memory*, for example, is a register-like memory that can be accessed in a single GPU clock cycle by all the threads within the same block. It enables the acceleration of spatially local operations. Arbitrary writing and reading of data with GPU DRAM (Dynamic Random Access Memory) – also denoted as device memory – is granted in the form of *global memory*, allowing a lot of data parallel algorithms previously difficult to map to or are inefficient to execute on GPUs to be implemented, e.g. histogram processing and frequency space transforms [8]. *Texture memory* and *constant memory* also reside in DRAM and are read-only; they are both accelerated by specific hardware and can be used to store input data which suits their access patterns.

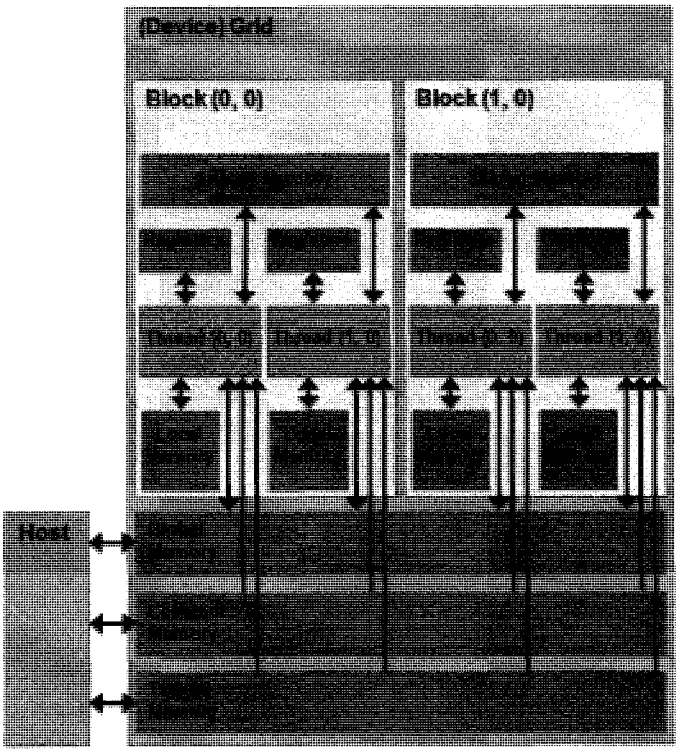


Figure 2.10: Memory programming model in CUDA.

A more detailed tutorial of CUDA can be found in [30]. Our effort in achieving real-time performance with the proposed algorithms is motivated by CUDA, with more details described in Chapter 4.

Chapter 3:
3D Adaptive Cost Aggregation for Slanted Surface Modeling and Sub-pixel Accuracy

In this chapter, a new approach is proposed to further improving the accuracy of original ASW method [53].

3.1 Motivation

According to the Middlebury stereo evaluation site, the best among all stereo algorithms have the feature of disparity plane fitting [20] [43] [50]. These approaches first over segment the image into small homogeneously-coloured regions, then apply plane-fitting technique to find candidate disparity planes for each segment. The optimal disparity plane assignment is determined using either local [43] or global [20] [50] optimization. Since the fitted disparity planes naturally provide sub-pixel disparity values, the scene can be reconstructed at a much finer level.

ASW uses a large aggregation window, whose size can be as big as 33×33. During aggregation the neighbourhood with the same disparity level is used. This approach can handle fronto-parallel surfaces that conform to the smoothness assumption suggested by Marr and Poggio [24]. Problem arises when the pixel actually lies on a slanted plane since the large window used in ASW means that

the neighbourhood has a bigger chance to include pixels of different disparity levels.

Inspired by the plane-fitting idea, a new cost aggregation approach that combines ASW with plane-fitting is introduced here. It features per-pixel non-fronto-parallel disparity plane modeling and performs ASW cost aggregation in the 3D cost volume along slanted planes.

3.2 Proposed System

The workflow of the system is described in Figure 3.1. Two disparity calculation passes are used. In the first pass, the algorithm computes an initial disparity map using the GPU-based ASW stereo matcher [16] [47]. Then, a disparity plane orientation (DPO) image which encodes the gradient of the disparity plane at each pixel is extracted using a simple least squares fitting approach. With estimated per-pixel DPO information, the newly designed 3D adaptive cost aggregation approach is used in the second pass for generating disparity results at sub-pixel accuracy. Finally, to refine the result, the disparity maps obtained for the two views are cross-checked to remove inconsistent disparity values, which are later filled in using a DPO-based hole-filling approach. A large window size (33×33 for example) with ASW is used to ensure the effectiveness of the slanted surface modeling. The disparity search space is also quantized at sub-pixel level to improve the accuracy of the disparity results. Each step is discussed in detail in the following sections.

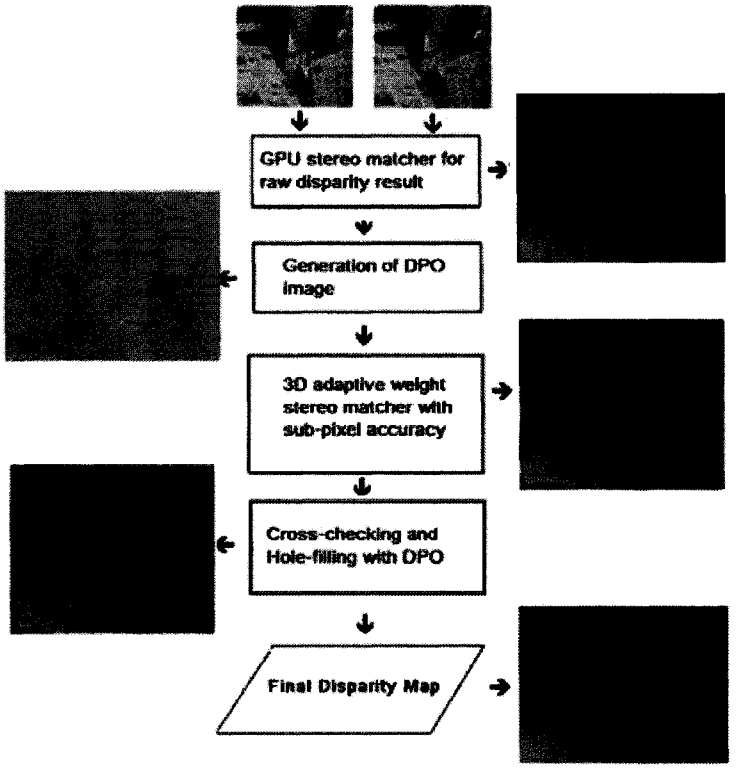


Figure 3.1: Workflow diagram of the proposed algorithm, with the input data, intermediate and final results.

3.2.1 Initial Disparity Map Generation

In this step, a raw disparity map of relatively good quality is desired at real-time speed. Wang *et al.* [47] ported several state-of-the-art local stereo algorithms onto GPU with proper simplification and found out that the simplified ASW gave the best result. Gong *et al.* [16] later made a more thorough survey and came up with a similar conclusion.

The simplified ASW equations used in both evaluation papers mentioned above are already introduced in Chapter 2 and restated here:

$$w(u,v,m,n) = exp\left(-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta g_{u,v,m,n}}{\gamma_g}\right)\right) \tag{3.1}$$

$$T(u,v,d) = \frac{\sum_{m=v,}^{r} w(u,v,m,0) \cdot C(u+m,v,d)}{\sum_{m=m,}^{r} w(u,v,m,0)} \tag{3.2}$$

$$AC(u,v,d) = \frac{\sum_{n=v,}^{r} w(u,v,0,n) \cdot T(u,v+n,d)}{\sum_{n=v+}^{r} w(u,v,0,n)} \tag{3.3}$$

The weight calculation (3.1) runs in exactly the same way as in the original ASW method. But instead of covering the whole aggregation window, only pixels on the same x or y axis with the pixel-of-interest have their weights calculated as shown in (3.2). Then a horizontal pass aggregates costs along the x-axis of the aggregation window for each pixel, and stores the results in a temporary 3D cost volume $T(u,v,d)$,. Finally a vertical pass aggregates along the y-axis for each pixel on $T(u,v,d)$ and stores results in the aggregated 3D cost volume $AC(u,v,d)$.

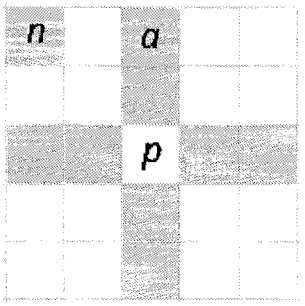


Figure 3.2: Weight calculation in the simplified ASW method.

The neighbour pixels that are not on the same axis with the pixel-of-interest will have indirect impacts while aggregating. In Figure 3.2, p is the pixel-of-interest, and n is the neighbour pixel that is not on the same axis as p. In the first

horizontal pass, n will exert its influence on a which shares the same axis with n and with p; then in the second pass the already aggregated a will pass along all the costs to p. $w(n_x, n_y, p_x, p_y)$ is computed as $w(n_x, n_y, a_x, a_y) \times w(a_x, a_y, p_x, p_y)$, which is fine when a and p have similar colours but is of poor quality when they differ much.

The resulting quality of disparity of the above mentioned method is satisfactory as raw input to our system. More importantly, it is fast. A single run with the *Tsukuba* dataset on NVIDIA 8800 GTS 512MB graphics card takes only about 5 ms (or 180 FPS speed), which is comparable to most of the other GPU-powered local stereo methods [16]. The ASW CPU implementation is claimed to take 1 minute using an AMD 2700+ processor [52]; even with current generation of CPU. Thus the speed-up is quite impressive. This module is also used in the real-time multiview stereo algorithm described in Chapter 5.

3.2.2 Disparity Plane Orientation Generation

The DPO image is essential to our proposed method since it encodes the gradient of the chosen disparity plane at each pixel location, which is later used in the final disparity computation.

To simplify calculations, here we ignore the foreshortening effect and assume that a plane in 3D world can be modeled by a plane in the 3D disparity space. The orientation of a given disparity plane is specified using the horizontal and vertical gradients (d_x, d_y) in the disparity space, where

$$d_x(u,v) = \frac{\partial \bar{D}(u,v)}{\partial u}, \quad d_y(u,v) = \frac{\partial \bar{D}(u,v)}{\partial v} \tag{3.4}$$

and $\bar{D}(u,v)$ is the unknown ground truth disparity map.

To estimate d_x and d_y from an inaccurate disparity map $D(u,v)$ obtained, a simple least squares fitting method is applied. For example to compute $d_x(u,v)$, we want to find a horizontal line that passes through $D(u,v)$ and gives the smallest weighted squared error. The weighted squared error between the data and the fitted straight line is defined as:

$$E = \sum_{k=-r}^{+r} z_{(u,v)}^{k} \Big(D(u+k,v) - \big(d_x(u,v)k + D(u,v)\big)\Big)^2 \tag{3.5}$$

where the weight function $z_{(u,v)}^{k}$ is used to suppress outliers. A simple step function is used here:

$$z_{(u,v)}^{k} = \begin{cases} 1 & if\ |D(u+k,v) - D(u,v)| \le 2 \\ 0 & otherwise \end{cases} \tag{3.6}$$

When E is the minimum, we have

$$\frac{\partial E}{\partial d_x(u,v)} = -2 \sum_{k=-r}^{+r} z_{(u,v)}^{k} k \Big(D(u+k,v) - \big(d_x(u,v)k + D(u,v)\big)\Big) = 0 \tag{3.7}$$

So $d_x(u,v)$ can be calculated as:

$$d_x(u,v) = \frac{\sum_{k=-r}^{+r} z_{(u,v)}^{k} D(u+k,v)k - D(u,v)\sum_{k=-r}^{+r} z_{(u,v)}^{k} k}{\sum_{k=-r}^{+r} z_{(u,v)}^{k} k^2} \tag{3.8}$$

The vertical gradient $d_y(u,v)$ is computed similarly. Using the *Venus* dataset as input, Figure 3.3 illustrates the generated DPO image, which keeps the horizontal and vertical gradient values in red and green channels respectively. The

left image is the raw disparity result; the middle image is the calculated DPO; and the right one has the ground truth of DPO.

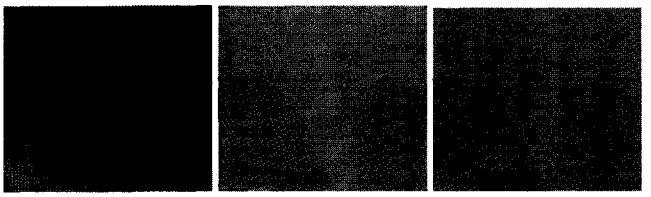


Figure 3.3: DPO generation.

3.2.3 3D Adaptive Cost Aggregation with Sub-pixel Accuracy

The original adaptive-weight cost aggregation approach assumes that all surfaces in the scene are fronto-parallel and matching costs are aggregated within 2D constant disparity planes. This assumption rarely holds in the real world, especially due to the large support window used — even when the slant is very small, the big neighbourhood span can still go through multiple disparity levels. In contrast, our approach performs aggregation in 3D disparity space along with the DPO estimated at different pixel locations.

Assume that the cost volume C holds the initial matching cost, where $C[u, v, k]$ gives the colour difference between pixels (u, v) in the left image and $(u - k, v)$ in the right image. Also assume that when d is non-integer, function $C(u, v, d)$ linearly interpolates between $C[u, v, \lfloor d \rfloor]$ and $C[u, v, \lceil d \rceil]$. The formula for the 3D adaptive-weight aggregation is as follows:

$$AC(u,v,d) = \frac{\sum_{m,n\in[-r,r]} \left(w(u,v,m,n) \cdot \left| C \begin{pmatrix} u+m, v+n, d + \\ md_x(u,v) + nd_y(u,v) \end{pmatrix} \right| \right)}{\sum_{m,n\in[-r,r]} w(u,v,m,n)} \tag{3.9}$$

where (u, v) is the pixel of interest; $w(u, v, m, n) = e^{-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta q_{u,v,m,n}}{\gamma_g}\right)}$ represents the weight of neighbour pixel $(u + m, v + n)$; $\Delta c_{u,v,m,n}$ and $\Delta q_{u,v,m,n}$ are the colour difference and the Euclidean distance between pixels (u, v) and $(u + m, v + n)$, respectively. γ_c and γ_g are the user defined parameters.

$AC(u, v, d)$ is the aggregated cost for assigning disparity hypothesis d to pixel (u, v), under the pre-computed DPO $d_x(u, v)$ and $d_y(u, v)$ at pixel (u, v). To generate disparity maps at sub-pixel accuracy, we step through disparity hypothesis d at 0.5 intervals.

After the aggregation process, the winner-take-all optimization is used to find the optimal disparity map:

$$D(u, v) = \text{agrmin}_{\text{d}}\, AC(u, v, d) \tag{3.10}$$

3.2.4 Cross-checking and Hole-filling

The above procedure is applied to both the left and right stereo images and the obtained disparity maps are cross-checked. The left pixel is mapped to the right image, according to the left disparity image; then the mapped right pixel is mapped back to the left image according to the disparity value in the mapped right pixel in the right disparity image. If the remapped left pixel is too far away from the original left pixel, say, by more than 2 pixels, then the original left pixel is labelled as a hole.

One interesting observation is that most of the holes are caused by occlusion. A quick fix to fill the hole is adopted here: each horizontal line is scanned to find the left and right immediate-neighbouring valid pixels for each hole and their disparity values are examined. Since the occluder has a larger disparity value than that of the occludee, the larger disparity value of the neighbours is chosen to fill the hole.

When propagating the chosen disparity, the horizontal gradient from DPO image is used to alter the chosen disparity value at a finer level so that the slanted surface is modeled.

3.3 Experimental Results

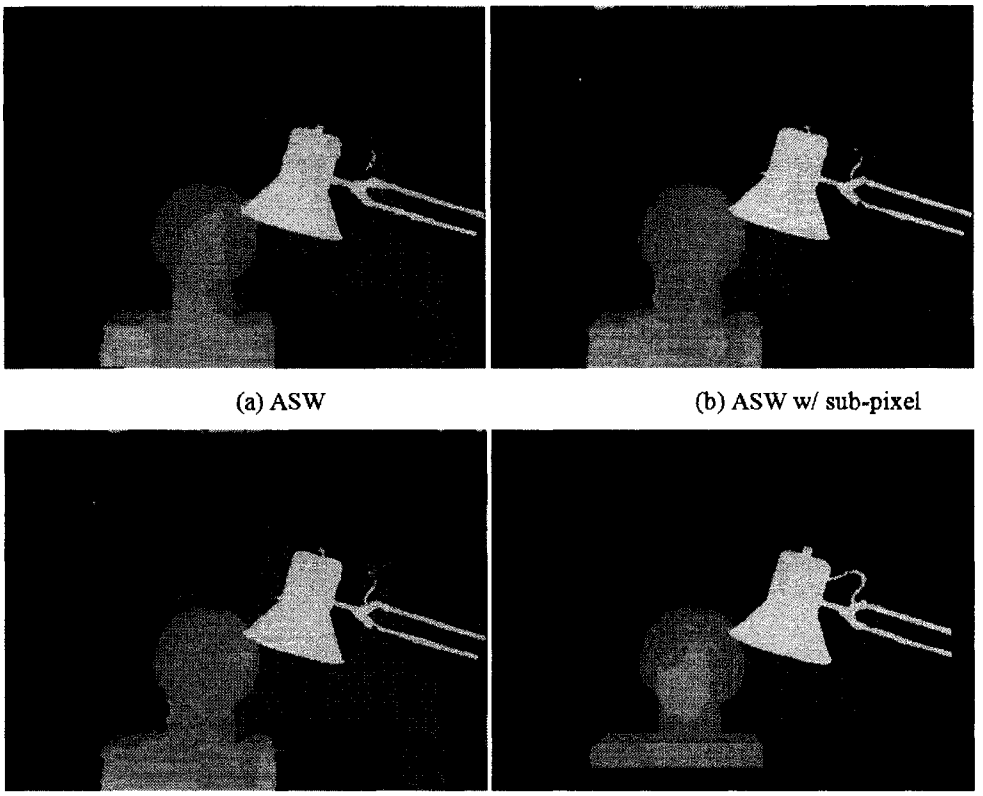
The proposed method was evaluated using the Middlebury testbed [26]. Some of the parameters used in the experiments follow the empirical choices in [16]. In particular, the support window size is set equal to 51×51 and the two parameters for support weight calculation: $\gamma_c = 19.6$ and $\gamma_g = 40$.

Figures 3.4~3.7 show results for different datasets. The original adaptive weight method, the implemented adaptive weight method with sub-pixel accuracy, and the new approach are all tested. Tables 3.1, 3.3, 3.5, and 3.7 give the statistical analysis on the performance of the algorithm, while Tables 3.2, 3.4, 3.6, and 3.8 have the statistical analysis information when sub-pixel error threshold is adopted.

From all the presented disparity results and statistical analysis, the following observations can be made:

1. For the Venus, Teddy and Cones datasets, where slanted surfaces are everywhere, the proposed slanted surface modeling approach effectively improves the results upon the original adaptive-weight stereo matcher. The improvement upon the Tsukuba dataset is minimal, since the presence of slanted planes appears very limited.
2. Direct incorporation of sub-pixel accuracy with the original adaptive weight stereo matcher does not provide any noticeable improvement.
3. Combining sub-pixel accuracy with slanted plane modeling gives very convincing results. When evaluated with a disparity error threshold of 1.0, the new algorithm is ranked 17th among all local and global optimization

approaches, as depicted in Figure 3.8; when a sub-pixel disparity error threshold of. 0.5 is used, it is ranked 4th, as illustrated by the snapshot taken from Middlebury vision stereo site in Figure 3.9. Other local stereo algorithms are underlined with red in both snapshots.



(a) ASW (b) ASW w/ sub-pixel

(c) Proposed (d) Ground Truth

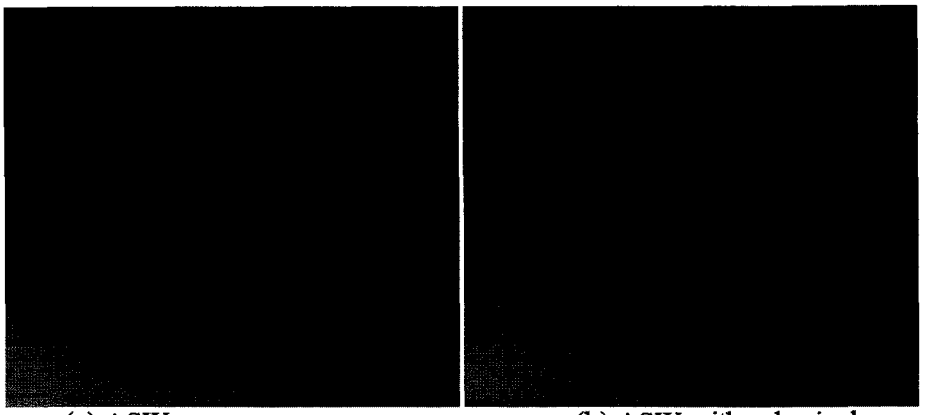
Figure 3.4: Results for the Venus dataset.

	Tsukuba		
Alg.	nonocc	all	disc
ASW	1.38 *17*	1.85 *15*	6.90 *17*
ASW.Subpixel	1.85 *23*	2.31 *21*	9.06 *25*
Proposed	1.79 *23*	2.30 *21*	8.79 *24*

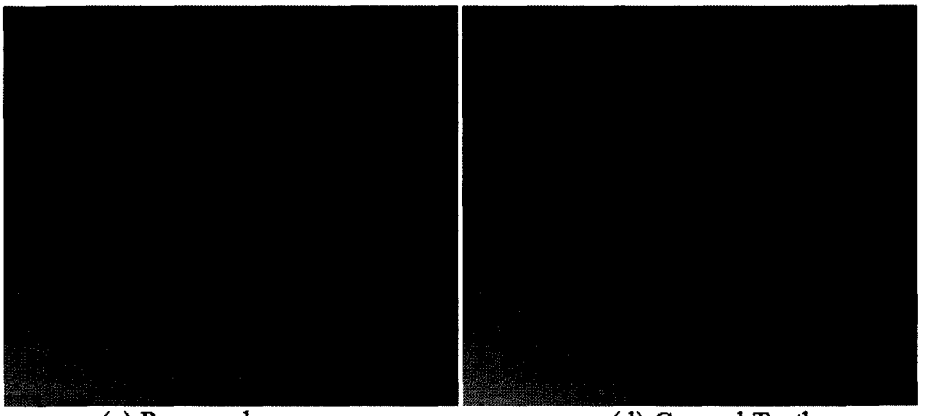
Table 3.1: Error rates evaluated with threshold of 1.0. Italic numbers are ranks. 'noonocc' column evaluates non-occluded areas, 'disc' column evaluates disparity discontinuity areas, and 'all' column evaluates every pixel in the disparity image.

	Tsukuba		
Alg.	nonocc	all	disc
ASW	18.1 *19*	18.8 *18*	18.6 *19*
ASW.Subpixel	9.60 *10*	10.2 *10*	14.8 *5*
Proposed	8.86 *9*	9.52 *7*	15.0 *6*

Table 3.2: Error rates evaluated with a threshold of 0.5. Italic numbers are ranks.



(a) ASW (b) ASW with sub-pixel



(c) Proposed (d) Ground Truth

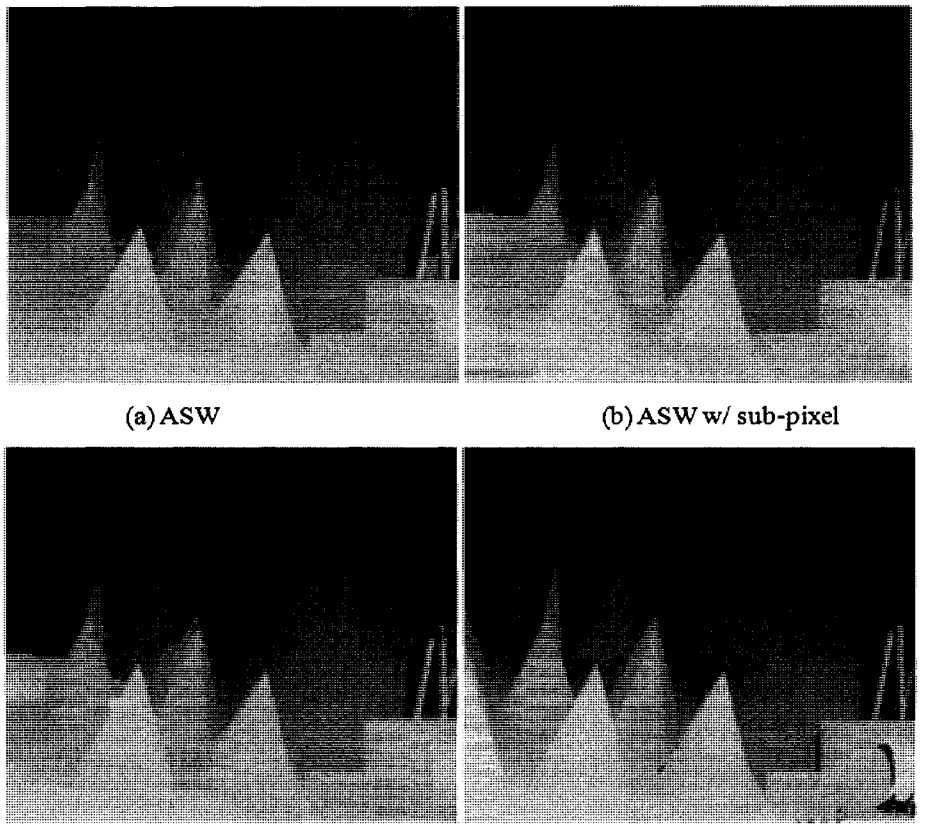
Figure 3.5: Results for the Venus dataset.

	Venus		
Alg.	nonocc	all	Disc
ASW	0.71 *18*	1.19 *19*	6.13 *19*
ASW.Subpixel	0.82 *21*	1.02 *18*	6.11 *19*
Proposed	0.30 *14*	0.54 *10*	3.63 *17*

Table 3.3: Error rates evaluated with a threshold of 1.0. Italic numbers are ranks.

	Venus		
Alg.	nonocc	all	disc
ASW	7.77 *18*	8.40 *20*	15.8 *19*
ASW.Subpixel	8.31 *21*	8.65 *21*	17.8 *24*
Proposed	2.99 *4*	3.29 *3*	8.17 *4*

Table 3.4: Error rates evaluated with a threshold of 0.5. Italic numbers are ranks.



(a) ASW (b) ASW w/ sub-pixel

(c) Proposed (d) Ground Truth

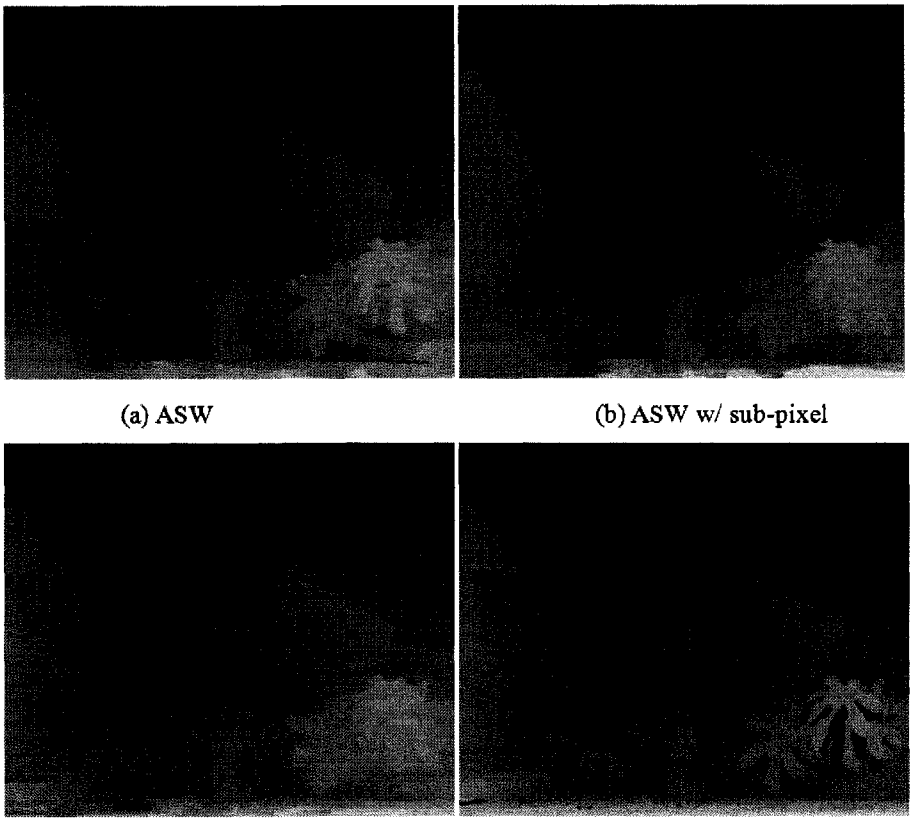
Figure 3.6: Results for the Cones dataset.

	Cones		
Alg.	nonocc	all	Disc
ASW	3.97 *18*	9.79 *14*	8.26 *6*
ASW.Subpixel	5.20 *26*	11.2 *23*	11.6 *22*
Proposed	3.72 *12*	9.27 *11*	9.70 *14*

Table 3.5: Error rates evaluated with a threshold of 1.0. Italic numbers are ranks.

	Cones		
Alg.	nonocc	all	disc
ASW	14.0 *24*	19.7 *23*	20.6 *19*
ASW.Subpixel	14.8 *27*	21.1 *26*	23.4 *26*
Proposed	8.81 *10*	15.1 *12*	17.5 *12*

Table 3.6: Error rates evaluated with a threshold of 0.5. Italic numbers are ranks.



(a) ASW (b) ASW w/ sub-pixel

(c) Proposed (d) Ground Truth

Figure 3.7: Results for the Teddy dataset.

	Teddy		
Alg.	nonocc	all	Disc
ASW	7.88 *19*	13.3 *19*	18.6 *23*
ASW.Subpixel	10.3 *29*	15.6 *27*	22.5 *31*
Proposed	7.11 *14*	8.45 *5*	17.5 *17*

Table 3.7: Error rates evaluated with a threshold of 1.0. Italic numbers are ranks.

	Teddy		
Alg.	nonocc	all	disc
ASW	17.6 *19*	23.9 *19*	34.0 *27*
ASW.Subpixel	21.1 *28*	27.3 *28*	37.4 33
Proposed	13.3 *10*	15.0 *2*	27.0 *10*

Table 3.8: Error rates evaluated with a threshold of 0.5. Italic numbers are ranks.

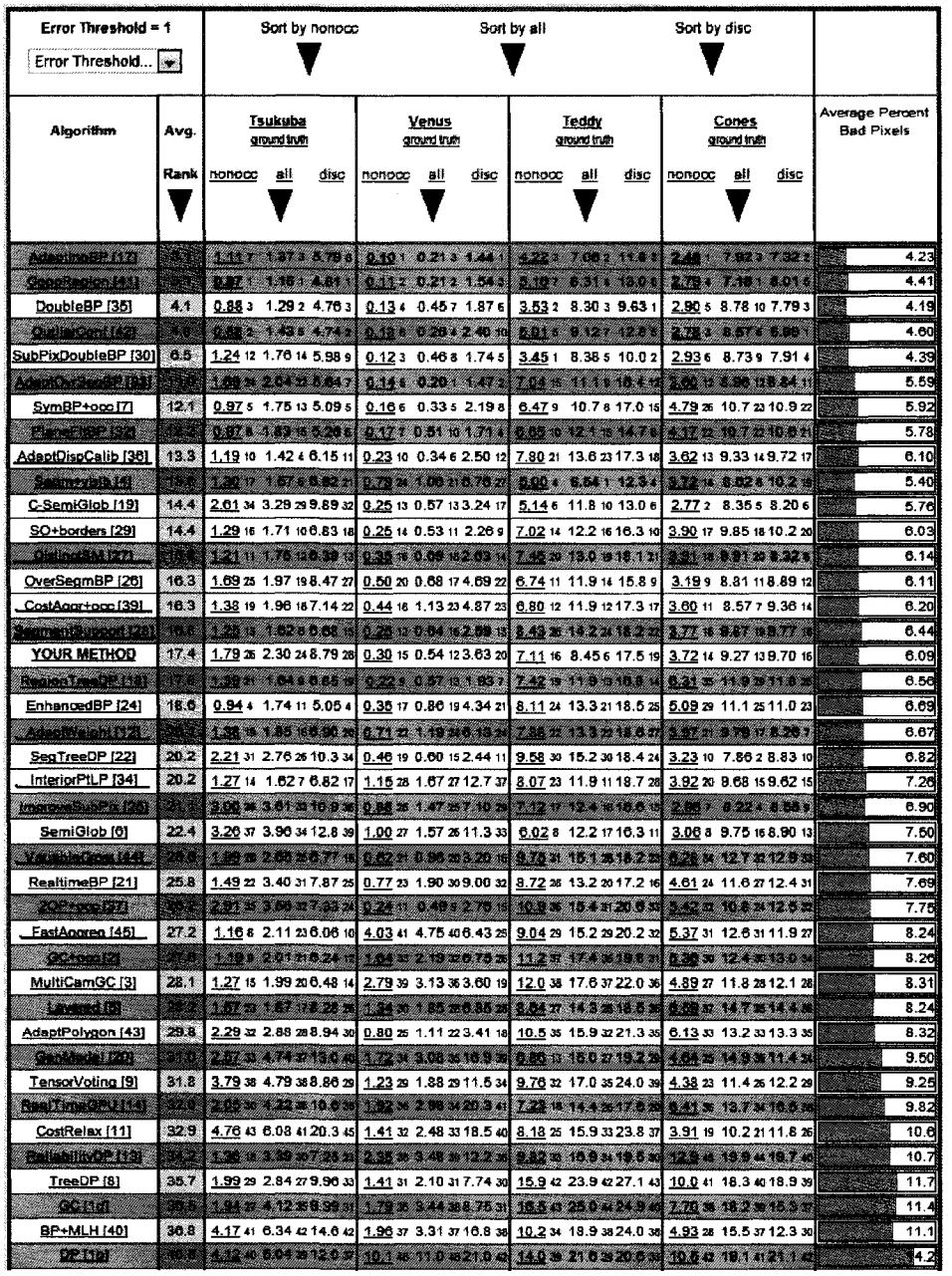
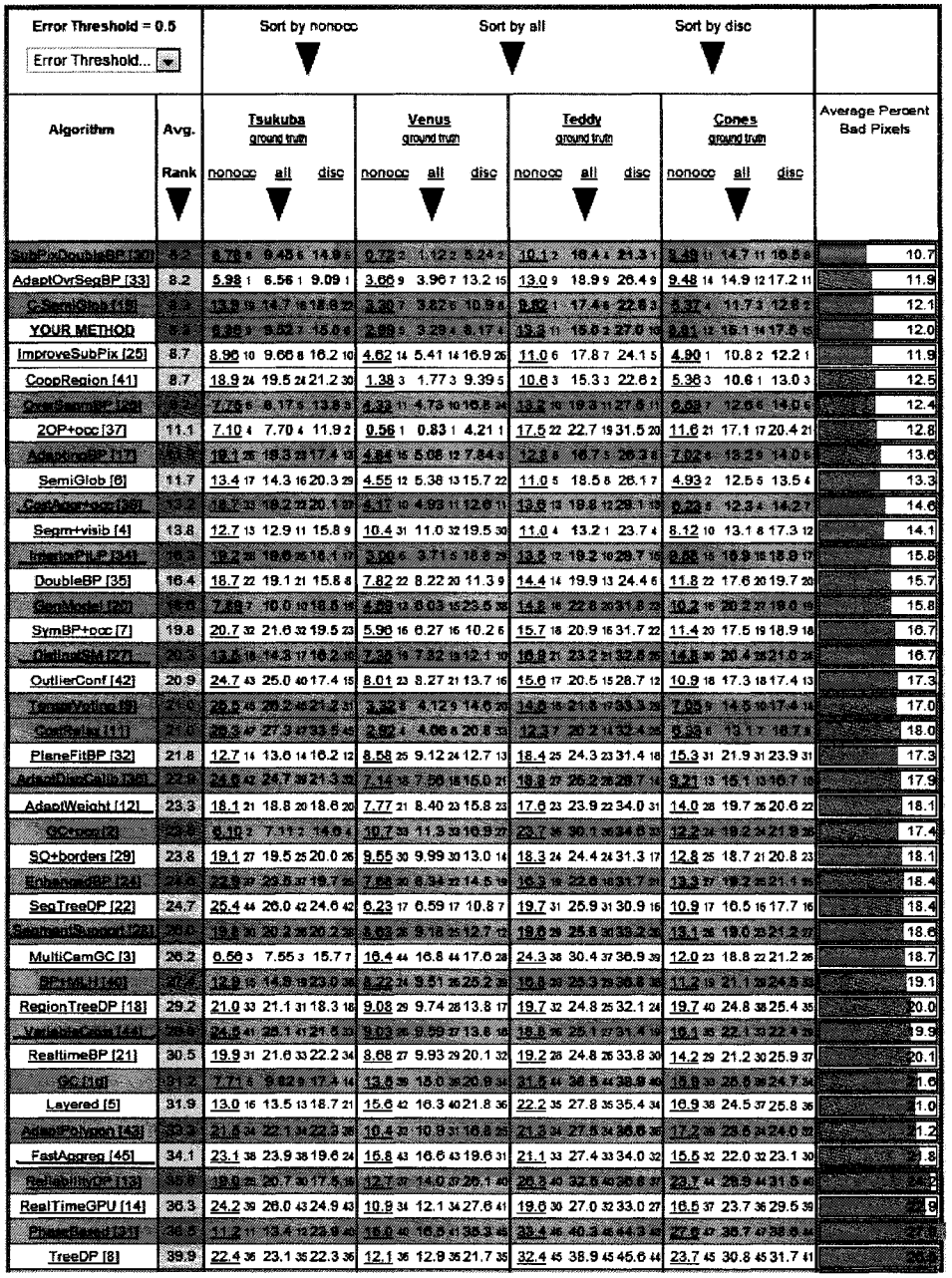


Figure 3.8: Ranking snapshot in Middlebury stereo vision site, with a disparity error threshold of 1. Algorithms underlined with red are other local stereo algorithms. The table is not complete as the list is long.



Error Threshold = 0.5 (Error Threshold...)		Sort by nonocc			Sort by all			Sort by disc			
Algorithm	Avg. Rank	Tsukuba ground truth nonocc	Tsukuba ground truth all	Tsukuba ground truth disc	Venus ground truth nonocc	Venus ground truth all	Venus ground truth disc	Teddy ground truth nonocc	Teddy ground truth all	Teddy ground truth disc	Cones ground truth nonocc	Cones ground truth all	Cones ground truth disc	Average Percent Bad Pixels
SubPixDoubleBP [30]	6.2	8.78 6	9.45 6	14.0 6	0.72 2	1.12 2	5.24 2	10.1 2	16.4 4	21.3 1	5.49 11	14.7 11	19.5 8	10.7
AdaptOvrSegBP [33]	8.2	5.98 1	6.56 1	9.09 1	3.66 9	3.96 7	13.2 15	13.0 9	18.9 9	26.4 9	9.48 14	14.9 12	17.2 11	11.9
C-SemiGlob [18]	9.5	13.8 19	14.7 18	18.6 22	3.38 7	3.82 6	10.9 8	9.82 1	17.4 6	22.6 3	5.37 4	11.7 3	12.6 2	12.1
YOUR METHOD	9.8	6.95 3	9.52 7	15.0 8	2.69 5	3.29 4	8.17 4	13.3 11	18.6 8	27.0 10	8.81 13	16.1 14	17.5 15	12.0
ImproveSubPix [25]	8.7	8.96 10	9.66 8	16.2 10	4.62 14	5.41 14	16.9 26	11.0 6	17.8 7	24.1 5	4.90 1	10.8 2	12.2 1	11.9
CoopRegion [41]	8.7	18.9 24	19.5 24	21.2 30	1.38 3	1.77 3	9.39 5	10.6 3	15.3 3	22.6 2	5.36 3	10.6 1	13.0 3	12.5
CostSamp+occ [39]	9.8	7.75 5	8.17 5	13.8 5	4.33 11	4.73 10	16.8 24	13.2 10	19.3 11	27.5 11	6.69 7	12.6 6	14.0 6	12.4
2OP+occ [37]	11.1	7.10 4	7.70 4	11.9 2	0.56 1	0.83 1	4.21 1	17.5 22	22.7 19	31.5 20	11.6 21	17.1 17	20.4 21	12.8
AdaptiveBP [17]	11.3	10.1 25	10.3 23	17.4 19	4.84 15	5.06 12	7.84 3	12.8 8	16.7 5	25.2 8	7.02 8	13.2 9	14.0 8	13.6
SemiGlob [6]	11.7	13.4 17	14.3 16	20.3 29	4.55 12	5.38 13	15.7 22	11.0 5	18.5 8	26.1 7	4.93 2	12.5 5	13.5 4	13.3
CostSamp+occ [39]	13.2	16.7 33	19.2 33	20.1 31	4.17 10	4.93 11	12.6 11	13.6 13	19.8 12	28.1 15	6.23 6	12.3 4	14.2 7	14.6
Segm+visib [4]	13.8	12.7 13	12.9 11	15.8 9	10.4 31	11.0 32	19.5 30	11.0 4	13.2 1	23.7 4	8.12 10	13.1 8	17.3 12	14.1
ImproPlUP [34]	15.3	16.2 28	16.6 26	16.1 17	3.00 6	3.71 5	16.8 25	13.8 12	19.2 10	29.7 16	8.66 12	19.9 16	18.8 17	15.8
DoubleBP [35]	16.4	16.7 22	19.1 21	15.8 8	7.82 22	8.22 20	11.3 9	14.4 14	19.9 13	24.4 6	11.8 22	17.6 20	19.7 20	15.7
GenModel [20]	16.6	7.89 7	10.0 10	18.8 16	6.89 13	8.03 15	23.5 36	14.8 16	22.8 20	31.8 23	10.2 16	20.2 27	19.6 18	15.8
SymBP+occ [7]	19.8	20.7 32	21.6 32	19.5 23	5.96 16	6.27 16	10.2 6	15.7 18	20.9 16	31.7 22	11.4 20	17.5 19	18.9 18	16.7
DistinctSM [27]	20.3	13.6 16	14.3 17	18.2 15	7.36 19	7.82 19	12.1 10	16.8 21	23.2 21	32.6 26	14.6 30	20.4 32	21.0 25	16.7
OutlierConf [42]	20.9	24.7 43	25.0 40	17.4 15	8.01 23	8.27 21	13.7 16	15.6 17	20.5 15	28.7 12	10.9 18	17.3 18	17.4 13	17.3
TensorVoting [9]	21.0	26.5 45	26.2 44	21.2 31	3.32 8	3.12 8	14.6 20	14.6 15	21.5 17	33.3 32	7.05 9	14.5 10	17.4 14	17.0
CostFilter [11]	21.6	29.3 47	27.3 47	33.5 48	2.82 4	4.06 8	20.8 33	12.3 7	20.2 14	32.4 25	6.33 5	13.1 7	16.7 8	18.0
PlaneFitBP [32]	21.8	12.7 14	13.6 14	16.2 12	8.58 25	9.12 24	12.7 13	18.4 25	24.3 23	31.4 18	15.3 31	21.9 31	23.9 31	17.3
AdaptDispCalib [36]	22.8	24.6 42	24.7 39	21.3 32	7.14 18	7.66 18	15.0 21	19.8 27	26.2 26	28.7 14	9.21 13	15.1 13	16.7 10	17.9
AdaptWeight [12]	23.3	18.1 21	18.8 20	18.6 20	7.77 21	8.40 23	15.8 23	17.6 23	23.9 22	34.0 31	14.0 28	19.7 26	20.6 22	18.1
GC+occ [3]	23.6	6.10 2	7.11 2	14.9 4	10.7 33	11.3 33	16.9 27	23.7 36	30.1 36	34.6 33	12.2 24	19.2 24	21.9 35	17.4
SO+borders [29]	23.8	19.1 27	19.5 25	20.0 26	9.55 30	9.99 30	13.0 14	18.3 24	24.4 24	31.3 17	12.8 25	18.7 21	20.8 23	18.1
EnhancedBP [24]	24.5	22.9 37	23.5 37	19.7 25	7.68 20	8.34 22	14.5 19	16.2 19	22.6 18	31.7 21	13.3 27	19.2 25	21.1 33	18.4
SegTreeDP [22]	24.7	25.4 44	26.0 42	24.6 42	6.23 17	6.59 17	10.8 7	19.7 31	25.9 31	30.9 16	10.9 17	16.5 16	17.7 16	18.4
SegmentBorder [23]	25.0	19.8 30	20.2 28	20.2 28	8.63 26	9.18 26	12.7 12	19.6 29	25.6 30	32.2 30	13.1 26	19.0 23	21.2 27	18.6
MultiCamGC [3]	26.2	6.56 3	7.55 3	15.7 7	16.4 44	16.8 44	17.6 28	24.3 38	30.4 37	36.9 39	12.0 23	18.8 22	21.2 26	18.7
SPTMLH [40]	26.4	12.9 10	14.9 15	23.0 36	8.22 24	9.51 28	25.2 38	19.8 28	25.3 29	36.8 38	11.2 19	21.1 29	24.5 38	19.1
RegionTreeDP [18]	29.2	21.0 33	21.1 31	18.3 18	9.08 29	9.74 28	13.8 17	19.7 32	24.8 25	32.1 24	19.7 40	24.8 38	25.4 35	20.0
VariableCross [44]	29.5	24.6 41	25.1 41	21.6 33	9.03 28	9.59 27	13.6 16	18.8 26	25.1 27	31.4 19	16.1 35	22.1 35	22.4 32	19.9
RealtimeBP [21]	30.5	19.9 31	21.6 33	22.2 34	8.68 27	9.93 29	20.1 32	19.2 26	24.8 26	33.8 30	14.2 29	21.2 30	25.9 37	20.1
GC [16]	31.2	7.71 8	8.62 8	17.4 14	13.6 35	15.0 36	20.9 34	31.5 44	36.5 44	38.9 40	19.8 39	26.8 39	24.7 34	21.6
Layered [5]	31.9	13.0 16	13.5 13	18.7 21	15.6 42	16.3 40	21.8 36	22.2 35	27.8 35	35.4 34	16.9 38	24.5 37	25.8 36	21.0
AdaptPolynom [43]	33.2	21.8 34	22.1 34	22.3 35	10.4 32	10.9 31	16.8 25	21.3 34	27.5 34	36.6 36	17.2 36	23.6 34	24.0 32	21.2
FastAggreg [45]	34.1	23.1 38	23.9 38	19.6 24	16.8 43	16.6 43	19.6 31	21.1 33	27.4 33	34.0 32	15.5 32	22.0 32	23.1 30	21.8
ReliabilityDP [13]	35.6	19.0 29	20.7 30	17.5 13	12.7 37	14.0 37	26.1 40	26.3 43	32.5 43	35.8 37	23.7 44	28.9 44	31.0 45	23.2
RealTimeGPU [14]	36.3	24.2 39	26.0 43	24.9 43	10.9 34	12.1 34	27.6 41	19.6 30	27.0 32	33.0 27	16.5 37	23.7 36	29.5 39	22.9
PhaseBased [31]	36.5	11.2 11	13.4 12	23.9 40	15.0 40	16.9 41	38.3 45	33.4 46	40.3 46	44.3 45	27.9 47	35.7 47	38.6 44	27.9
TreeDP [8]	39.9	22.4 36	23.1 35	22.3 36	12.1 36	12.9 35	21.7 35	32.4 45	38.9 45	45.6 44	23.7 45	30.8 45	31.7 41	26.2

Figure 3.9: Ranking snapshot in Middlebury stereo vision site, with a disparity error threshold of 0.5. Algorithms underlined with red are other local stereo algorithms. The snapshot is not complete as the list is long.

3.4 Summary

In this chapter a novel local stereo algorithm combining ASW with plane fitting is introduced. The least squares fitting method is used for robustly estimating per-pixel DPO information, which is later used to guide cost aggregation along slanted surfaces in 3D cost volume. The experimental results show that the new approach produces better disparity maps than the original ASW algorithm and is comparable with other local and global algorithms. Compared to existing plane fitting based algorithms [20] [43] [50], the new approach does not require *a priori* image segmentation and is easier to implement in current generation of GPU.

Chapter 4:
Near Real-time Adaptive Support Weight Cost Aggregation with CUDA

This thesis tries to advance local stereo matching from both aspects of quality of results and processing speed. Whereas the previous chapter presents a novel ASW based algorithm that gives results of better quality, this chapter focuses on how to accelerate the original ASW algorithm through implementing it on GPU using the CUDA programming language.

4.1 Motivation

The ASW algorithm proposed in [52] has been proven to be among the best local stereo matching algorithms [45], but the good performance comes with a high computational cost. The CPU version, as reported by Yoon and Kweon, takes one minute on an AMD 2700+ machine for the *Tsukuba* dataset when a 33×33 support window is used [52]. Being a local method and bearing great amount of parallelism, it offers room for improvement of speed.

Some efforts have already been made along this direction. As described in Section 3.2.1, Wang *et al.* [47]and Gong *et al.* [16] both use a simplified weight equation to push the algorithm to the real-time performance boundary. In particular, in the local window, only weights for pixels along the same vertical and horizontal scanlines as the pixel-of-interest are calculated. The weights for the

rest of the pixels are approximated, as described in Section 3.2.1. The speed-up is achieved at the cost of disparity map quality; as evaluated in both papers, the number of bad pixels increases by 30%~400% for the four datasets compared to the original CPU-based ASW approach.

Thus, the porting of the ASW cost aggregation method onto GPU without loss of stereo result quality is worthwhile and is the main goal of this chapter.

4.2 Simple Implementations on CUDA

4.2.1 Direct Porting to GPU

The computations conducted for every pixel is tedious but straightforward. For each pixel, a big square-shaped neighbourhood is examined with weights for pixels inside generated and the weighted cost aggregated. This process is repeated for each disparity level to give the aggregated cost volume. Then the winner-take-all method decides the final disparity result for each pixel. The CPU code can be directly ported onto the GPU using the CUDA language. The pseudo-code can be found in Appendix I.

Performance and Analysis

The above straightforward code migration does bring efficiency improvement, just as most of the implementations with enough parallelism will always benefit from running on GPU. For example, for the *Tsukuba* dataset, which has a dimension of 384×288, a disparity range of 12, and an aggregation window of size 33×33, the running time is a little over 1 second using kernel blocks of size 16×16 on a single NVIDIA 8800 GTS 512MB graphics card. The speed up is huge compared to 3 minutes using the un-optimized CPU version on the same machine with 2.2 GHz AMD Opteron 2214 processor. Some minor loss of accuracy is present that we believe can be attributed to the less accurate floating point mathematical operations on GPU. Figure 4.1 has the result for CPU ASW

implementation running on the Tsukuba dataset, as well as the error map evaluated in Middlebury vision site; black pixels mean bad matches. The result of the GPU ASW implementation is presented in Figure 4.2. Table 4.1 has the numerical comparison. Note that no pre-processing or post-processing is used in both implementations, thus the evaluation results are not exactly the same as the ones reported in the original ASW paper [52].

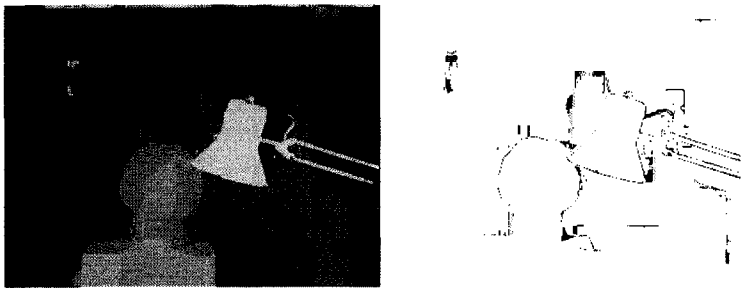


Figure 4.1: CPU ASW result and error map for the Tsukuba dataset.

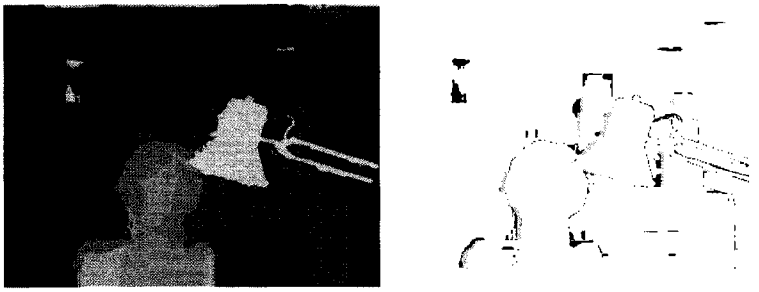


Figure 4.2: GPU ASW result and error map for the Tsukuba dataset.

	Tsukuba		
Alg.	nonocc	all	Disc
CPU ASW	1.91	2.29	8.75
GPU ASW	2.04	3.11	7.66

Table 4.1: Error rates evaluated with threshold 1.0.

The problem with this simple migration is obvious. There are too many texture accesses in a single kernel run, $2 \times szAggrWin^2 + 2$ to be exact. As noted in [30], one key to achieve the efficient use of GPU using CUDA code is to have a high computation to memory access ratio. GPU devotes more of its transistors in data processing than to data caching and flow control. Thus the memory access latency is high — 400~600 clock cycles of latency for a single memory read compared to 4 clock cycles for a floating point addition [30]. The way GPU works around this is by saturating the GPU with thousands of kernels so that when some kernels are idling waiting for the completion of memory accesses, other kernels can be switched in to keep the GPU busy. In the above code, however, the computation to memory access ratio is nearly 1 as almost every step of the calculation needs to read something new from the device memory. To increase this access ratio, either the number of memory accesses has to be lowered or the number of computations in each kernel has to be increased.

One side note on the memory access pattern with CUDA is that, the global memory accesses have to be coalesced to improve on speed. When the kernel is running on the GPU, there are always 16 kernels that run physically together on each multi-processor which has 16 stream processors. These 16 kernels are regarded as a kernel half-warp, or a kernel batch. For a line of code that performs memory operation, if these 16 kernels are accessing consecutive blocks of device memory, their accesses can be grouped into one single memory read, *i.e.* coalesced; otherwise these 16 memory accesses have to line up and proceed sequentially, which obviously increase the overall access time. With the above code, memory coalescing is automatically achieved as the 16×16 kernel block setup ensures that the kernel batch always access 16 consecutive addresses in $cv[][]$. Texture *targetIm* also resides in GPU DRAM, and its coalescing is automatically achieved through the texture buffering hardware if locality in texture accessing pattern is satisfied.

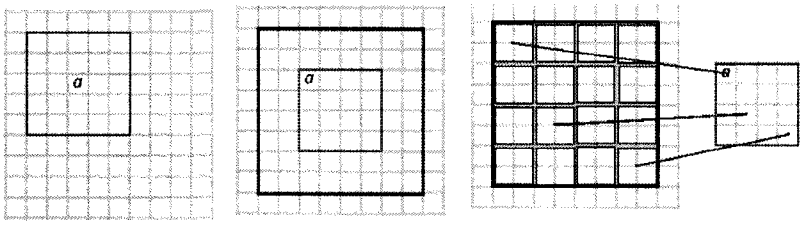
4.2.2 Improved Direct Porting to CUDA

CUDA exposes the nice features of the shared memory architecture of the G80 and later graphics cards to programmers. Shared memory is a register type memory which has single clock cycle latency, and can be accessed by all kernels within the same block with proper synchronization. Every multi-processor chip has 16K Bytes of shared memory. Thus, the size is limited and no abuse is possible; but a clever use of shared memory can almost always make the program run faster.

One straightforward improvement is to process through all disparity levels in a single kernel. The weights calculated are identical for all disparity levels for all pixels, and when the processing of different disparity levels is done in separate passes, these weights have to be re-calculated. By processing all disparity levels in one single kernel, many memory accesses and calculations can be saved. To achieve the above, in each kernel an array of matching costs of size *szRangeDisp* is allocated to sum up the weighted costs when walking through the neighbourhood. This array cannot reside in the register memory, as the register memory is scarce meaning that every kernel has only a handful of registers to use. There are 8K bytes of registers on each multiprocessor to be exact, thus to fit at least one kernel block of size 16×16 onto the multiprocessor at most 32 registers can be used. If this limit is exceeded, CUDA will resort to DRAM memory, which is hundreds of times slower than register memory and results in a major drop in performance. Hence, these arrays must be explicitly allocated in shared memory, which is as fast as register memory and bigger in size.

Another possible improvement uses the fact that neighbour kernels often scan through a similar neighbourhood when aggregating. Thus, for a block of pixels, if a superset of all neighbours for every pixel is pre-fetched into the shared memory, later on the weight calculations can get pixel information from the shared memory pool instead of from the GPU DRAM. Figure 4.3 depicts how this pre-fetching is

done. Suppose a 5×5 aggregation window is used, (a) shows pixel a and its neighbourhood (the red square). The CUDA kernel block is of size 4×4, as shown in (b) with the green square, with pixel a and all the other pixels belonging to the same block. The pre-fetched region into the shared memory is shown in (b) as the black square; it has a size of 8×8 and contains all the pixels needed to process the aggregation for the whole block containing a. Without the pre-fetching scheme and shared memory utilization, pixel a needs to look into 25 pixel positions as the red square embraces in (a). When the pre-fetching scheme is used, the whole block shares the load of reading in the 8×8 patch. Thus each pixel of the block only needs to read 4 memory positions. As depicted in (c), each pixel in the block is responsible for prefetching a 2×2 red squares, with the corresponding kernels and 2×2 squares connected by blue lines (only a few of which are drawn for the purpose of illustration). Thus the high-latency DRAM accesses in each kernel decrease from 25 to 4 in this specific scenario. In fact, a combination of an aggregation window size $(2W + 1) \times (2W + 1)$ and a kernel block size $K \times K$ sees the DRAM access count change from $(2W + 1) \times (2W + 1)$ to $\left\lceil \frac{(2W+K)^2}{K^2} \right\rceil$ with this scheme. In the setup we adopt (window size 33×33 and kernel block size 8×8), the high-latency DRAM access number each kernel decreases from 33×33 to just 25, thus a speed-up in performance is expected.



(a) (b) (c)

Figure 4.3: How to use shared memory to reduce latency.

Performance and Analysis

The pseudo-codes can be found in Appendix II. Due to the improved memory access pattern, an increase in performance was expected. The running time for both algorithms are improved to 300 ms as compareed to the 1s performance from the direct porting implementation.

But there are still some hardware limitations that deny a better performance for both algorithms. First of all, the number of kernels within each block has to be big enough to achieve the best latency hiding and GPU saturation. The recommended configuration is 256 kernels or more for each block. Let's assume a 16×16 block setup is used along with a 33×33 aggregation window in algorithm AII.2. Then *targetImPatch* and *cvPatch*[] both need a size of 48×48. With *targetImPatch* being 32 bit RGBA colour type and *cvPatch* holding 32bit floating point values, the total amount of shared memory required is $48 \times 48 \times 4 \times 2 = 18432$ bytes, which is too big to fit in the 16K byte shared memory, not to mention that part of the shared memory is reserved for other uses like storing kernel function parameters. So either the size of the aggregation window or the kernel block has to decrease. Unfortunately, both methods have their stumbling blocks: if the size of the aggregation window is smaller, the quality of the final disparity result will suffer; on the other hand, if the kernel block size is shrunk, the memory loading burden for each kernel is increased and also the saturation of GPU is reduced.

Secondly, several blocks have to be resident concurrently on a GPU multiprocessor if a higher saturation of GPU resource is intended. The same reason explains why a big block size is preferred. The contexts and resources are pre-allocated for every block, so that when some blocks are idling, other blocks can be switched in without much overhead. This scheme of using CUDA means that resources like register and shared memory have to be split among several blocks. In fact, given the kernel block size, number of registers each kernel uses,

and the consumption of shared memory for each kernel block, the GPU saturation level can be calculated using the CUDA occupancy calculator [29]. Since it is already hard for both algorithms to fit one single block of kernels into a GPU multi-processor, this programming pattern for better GPU utilization is again not met.

Thirdly, no matter how good the memory access scheme is, the actual number of computations is still intact provided that the aggregation window of the same size is used. Each kernel still needs to scan through the neighbourhood, compute all the weights, and aggregate all the costs over the neighbourhood.

4.3 Segmentation Driven Adaptation

Direct porting the adaptive weight cost aggregation method onto GPU poses problems that cannot be easily addressed using current generation of GPUs. Thus a better method has to be developed.

The segmentation based ASW aggregation method [44] achieves a great improvement over the original ASW aggregation approach. Given the segmentation information, weights for neighbour pixels that are in the same segment as the pixel-of-interest are set to 1.0, as indicated in equation (4.1). This simple change is demonstrated to be effective in dealing with depth borders, low-textured regions, high-textured regions, and repetitive patterns. The actual aggregation, however, is still performed on a pixel by pixel basis as shown in equation (2.5).

$$w(u,v,m,n) = \begin{cases} 1.0, & if\ Seg(u,v) = Seg(u+m, v+n) \\ exp\left(-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta g_{u,v,m,n}}{\gamma_g}\right)\right), & otherwise \end{cases} \tag{4.1}$$

If segments are the primitives to be processed in the segmentation based approach, all the weight calculation and cost aggregation can be done in a similar manner as the pixel primitive implementation. Since a segment is effectively represented by a single pixel colour, the segment-to-segment weights can be calculated as pixel-to-pixel weights with the proximity term being the Euclidean distance between segment centroids. Equation (4.2) has the idea demonstrated. *seg* is the segment-of-interest while *nseg* is a segment within neighbourhood. *seg.cen* denotes the centroid of segment *seg*. Equation (4.3) shows how the cost aggregation is done: for the segment-of-interest *seg*, a search within its neighbourhood *N(seg)* proceeds and finds all the segments within this neighbourhood; then with the segment-to-segment weight of *seg* and some neighbour segment *nseg*, the collective matching costs over area *nseg* at disparity *d* is summed into the final cost. Following this fashion, the pixel representation is completely replaced with the segment representation. Later on the winner-take-all disparity selection also chooses the best disparity on a per segment basis.

$$w(seg, nseg) = exp\left(-\left(\frac{\Delta c_{seg.color,nseg.color}}{\gamma_c} + \frac{\Delta g_{seg.cen,nseg.cen}}{\gamma_g}\right)\right) \tag{4.2}$$

$$AC(seg, d) = \frac{\sum_{nseg \in N(seg)} \begin{pmatrix} w(seg, nseg) \\ \cdot C(nseg,\ d) \end{pmatrix}}{\sum_{nseg \in N(seg)} w(seg, nseg)} \tag{4.3}$$

Since the number of segments is smaller than the number of pixels by a big margin, the computations needed to get the weights and aggregate costs are reduced significantly, which cannot be achieved by implementations mentioned in Section 4.2. This speed-up is achieved by sacrificing a bit of accuracy in the support weight, as the distance between segment centroids may not fully capture the various distances between different pixel-to-pixel pairs.

The nicest thing about this new proposed approach is that it can be fully ported onto GPU. Although the current generation of GPUs is not designed to support complex data structures, the quad-tree data structure is simple and efficient enough to be implemented on GPUs. The quadtree-based segmentation, though not as good in segmentation quality as other alternatives like watershed segmentation or mean-shift segmentation, has the perfect square shape of segments that is crucial for efficient GPU implementation, since operations like summing over a square segment is much easier to perform in a kernel than an arbitrarily shaped segment.

4.3.1 Quadtree Segmentation

Quadtree segmentation adopted here has two passes. The first pass, UpMerge, progresses in a bottom up fashion building a pyramid structure. Each level of the pyramid is half of the previous level in both dimensions. When building the current level, the corresponding four pixels of the previous level are evaluated to determine whether merging them is possible or not. Every pixel is associated with a flag: validSeg. If any of these pixels is flagged as not validSeg, then no merging occurs; if all four pixels can be merged, but their standard deviation is larger than a certain threshold c_0 meaning that they are too different to be included in the same segment, then merging again fails. Otherwise, merging succeeds and the average colour value is stored in the merged pixel position, with the validSeg flag set to true while the validSeg flags for the four lower level pixels are set to false. This is illustrated in Figure 4.4. The four pink pixels correspond to children of the red pixel in the upper level, while the four pale green pixels are merged into the green pixel if possible.

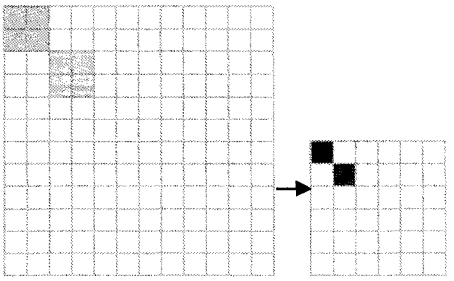
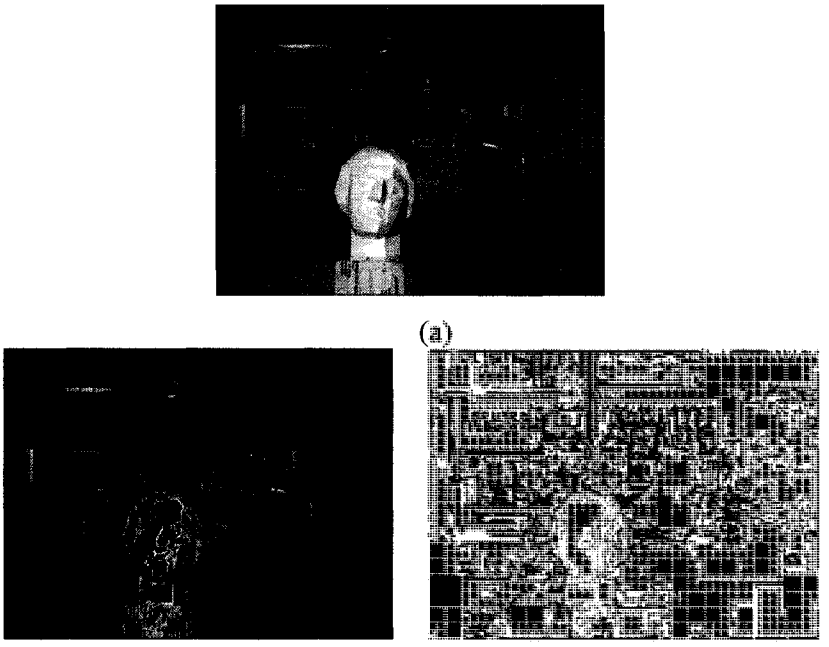


Figure 4.4: Quadtree segmentation, UpMerge pass.

The next pass is called Output, in which each level of the pyramid is examined and those pixels with a true validSeg flag are output into the segmentation image. In the segmentation image, only those pixels that are at the center of a valid segment contain meaningful RGBA values, in which the RGB channels store the segment average colour and the Alpha channel keeps the size of the segment. These pixels are denoted as segment pixels. The segmentation result is illustrated in Figure 4.5, where (a) shows the original image, (b) only shows segment pixels, and (c) add proper borders to segments.



(a)

(b) (c)

Figure 4.5: Quadtree segmentation result, with σ threshold = 10/255.

The neighbourhood traversal of a segment, as included in equation (4.3), is done by scanning through the neighbourhood window in the segmentation image and identifying those valid segment pixels within the neighbourhood.

4.3.2 Compact Segmentation Image

To search neighbour segments, scanning in the segmentation image is the better option. But launching kernels for every pixel of the segmentation image is a wasteful move, since only a small percent of kernels are created for valid segment pixels. The rest of the kernel will be completely idling during execution.

This waste of resources can be avoided by compressing the original sparse segmentation image into a compact segmentation image. Every single pixel in the compact segmentation image is a valid segment; therefore launching kernels for every pixel in the compact segmentation image will make all kernels busy. One of the scan primitives on GPU [38] named compact does exactly what is needed here, and fortunately most of the scan primitives have been efficiently implemented and exposed in CUDPP library [8].

4.3.3 Implementation

Algorithms 4.1, 4.2 and 4.3 have routines for the quadtree segmentation.

Algorithm 4.1

```
procedure UpMerge
  in: uchar4 pyramidLevel[n], float threshold
  out: uchar4 pyramidLevel[n+1]
begin
  for each node position on pyramidLevel[n+1] in parallel
    a, b, c, d ← four corresponding nodes in pyramidLevel[n]
    if σ ( a, b, c, d ) < threshold && validSeg flags for a, b, c, d are true then
      nodeValue.RGB = avg( a.RGB, b.RGB, c.RGB, d.RGB)
      nodeValue.segValid = true
      set node entry at pyramidLevel[n] to nodeValue
      set validSeg flags of a, b, c, d to false
```

else then
nodeValue.segValid = false
set node entry at *pyramidLevel*[n] to *nodeValue*
end

In procedure UpMerge defined in Algorithm 4.1, the standard deviation σ is computed over three colour channels by equation (4.5).

$$\sigma = \sqrt{\frac{1}{N}\{\sum_{i=1}^{N}(x_i.R - \bar{x}.R)^2 + \sum_{i=1}^{N}(x_i.G - \bar{x}.G)^2 + \sum_{i=1}^{N}(x_i.B - \bar{x}.B)^2\}} \tag{4.5}$$

Algorithm 4.2

---

procedure Output
in: uchar4 *pyramidLevel*[*n*]
out: uchar *segmentationImage*
begin
for each node position on *pyramidLevel*[*n*+1] **in parallel**
if *validSeg* of current node is true **then**
value.RGB = RGB value of current node
value.A = window size at current pyramid level
Set the corresponding pixel in *segmentationImage* to value
end

Procedure Output in algorithm 4.2 generates partial segmentation image by examining current level of the pyramid. Carrying out this procedure for every level of the pyramid gives the complete segmentation image. The window size at the current pyramid level is decided by n^2. The corresponding segment pixel position in segmentation image of the current node is decided by both its node position within the current pyramid $p(x, y)$ and the current pyramid level n, making sure it sits right in the middle of the segment. The segmentation image generated this way can ensure that the correct Euclidean distance between two

segments can be calculated by calculating the distance between their corresponding segment pixels.

Algorithm 4.3

```
procedure ProcessQuadTree
   in:  texture targetIm, float threshold
  out: uchar segmentationImage
begin
   copy targetIm into pyramidLevel[0] with padding
   for i=0 to k
      UpMerge( pyramidLevel[i-1], threshold, pyramidLevel[i] )
   for i=0 to k
      Output( pyramidLevel[i], segmentationImage )
end
```

In procedure ProcessQuadTree of algorithm 4.3, the padding is to make sure that at the base level of the pyramid, the size of the image is a power-of-2 so that the even splitting on each level of the quad tree can be achieved. And k is decided by the size of the biggest segment allowed, which is equal to 2^k to be exact. The output is a segmentation image described in Section 4.4.1.

Algorithm 4.4

```
procedure ProcessCostAggregationQuadTree
   in:   float cv[][], float threshold, int dispRange, texture targetIm
  out:  float cv[][]
begin

   ProcessQuadTree(targetIm, threshold, segmentationImage)
   for i=0 to dispRange
      for each pixel position p of segmentationImage in parallel
         if p is a valid segment then
            sum costs in cv[i] over its segment pixels and store the summed value
            into acv[i]

   Compact( segmentationImage, compactSegmentationImage )
```

```
for i=0 to dispRange
  for each pixel position of compactSegmentaionImage in parallel
    cost ← 0
    seg ← current pixel in compactSegmentationImage
    for each neighbour segment nseg of current segment
      wt ← calculate weight with seg and nseg information
      cost ← wt*acv[i][position of nseg]
    set cv[i][position of seg] to cost
end
```

Procedure ProcessCostAggregationQuadTree in Algorithm 4.4 makes use of the quadtree segmentation routine and carries out actual cost aggregation.

Note that when the segmentation image is used instead of the compact segmentation image, the search of neighbour segments can be coordinated for the whole thread block so that memory coalescing is achieved and code branching is avoided. Figure 4.6 illustrates the detailed traversal scheme. Pixels in pink comprise of a kernel block of size 4×4, while green pixels are neighbourhood that are pre-fetched into the shared memory. A spiral route is used in the search, ordered by the numeric labels of the pixels. Each pink pixel follow the exact same route, thus in every step of the kernel code they will take the same branch. In a compact segmentation image, the above mentioned locality within the thread blocks is damaged. Therefore, every kernel has to follow its own spiral path.

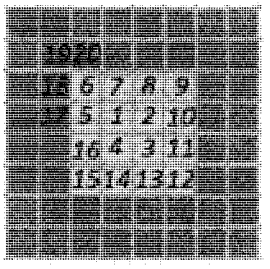


Figure 4.6: Search for neighbour segments.

4.3.4 Performance and Analysis

With the help of quadtree segmentation, the running speed now settles at around 100ms per frame for the *Tsukuba* dataset on a single NVIDIA 8800 GTS 512MB graphics card, with a disparity level of 12 and aggregation window of size 33×33. The kernel block is of size 16×16. The standard deviation threshold used in quadtree segmentation is set at 10. The comparison of results between segmentation driven ASW and the original ASW is given in Figure 4.7. The left image is the result from the segmentation driven ASW method, and the right one comes from the original ASW method. As you can see, some block shaped pixel patches are present throughout the disparity map, since the disparity choices are uniform within a segment. A numerical evaluation is given in Table 4.2, suggesting that not much change in disparity result quality happens. Therefore, a near real-time performance for ASW on GPU is achieved, without much loss of accuracy.

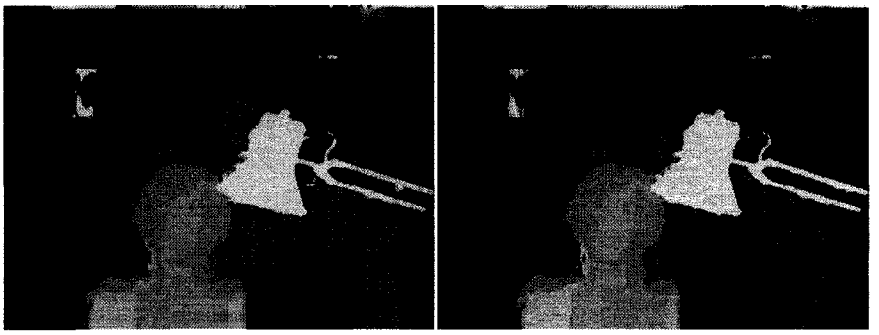


Figure 4.7: Disparity results for segmentation driven ASW and original ASW.

	Tsukuba		
Alg.	nonocc	all	disc
QT GPU ASW	2.00	3.01	7.36
GPU ASW	2.04	3.11	7.66
CPU	1.91	2.29	8.75

Table 4.2: Error rates evaluated with threshold 1.0.

With the standard deviation threshold of quadtree segmentation set lower, the segmentation image contains fewer segment pixels, thus the speed of the whole algorithm can improve even more. For a threshold of 30, the speed of the algorithm can reach 20 FPS. But the quality of the final result suffers as less accurate segmentation result is used to assist the aggregation. Figure 4.9 gives the comparison of results when different standard deviation threshold is applied. The first row has the disparity result and the segmentation image when threshold = 10, the second row uses threshold of 20, and the last row 30. The deterioration of quality is obvious when a lower threshold is used.

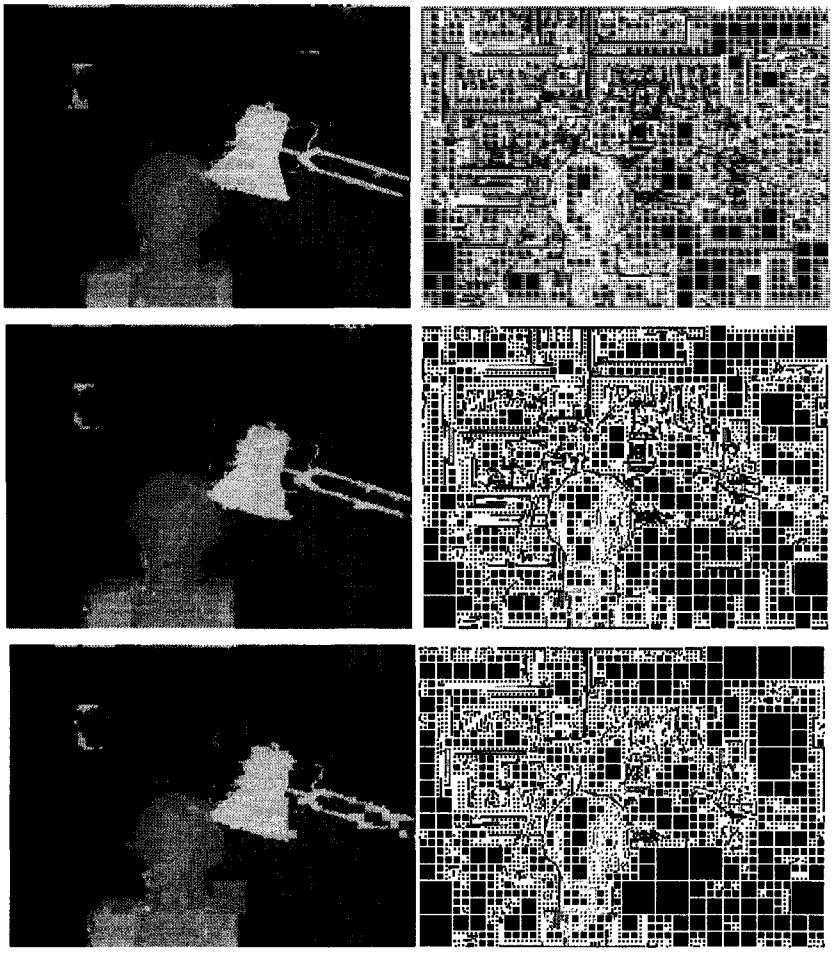


Figure 4.8: Disparity results and segmentation images when different thresholds are adopted.

4.4 Summary

This chapter presents several attempts on porting the original ASW onto GPU without loss of quality. The first one ports CPU code line-by-line directly into CUDA GPU implementation, the second one tries to optimize the directly GPU implementation with the help of CUDA features like shared memory. Both have hardware hindrance and cannot quite achieve good performance. Finally, the new quadtree driven ASW stereo algorithm improves performance by incorporating segmentation information into cost aggregation. As a result, near real-time speed is achieved without much loss of accuracy in the results.

Chapter 5:
Multi-view Stereo using Adaptive Weight and Parzen Window

The previous two chapters dedicate on improving quality or speed for binocular stereo matching. The occlusion problem however is not addressed due to its well-known limitation to binocular stereo. In this chapter, a novel real-time sparse multi-view algorithm is presented for better handling occlusions [54].

5.1 Motivation

Dense multi-view stereo algorithms can reconstruct the 3D model of a scene with high accuracy. But they are not yet suitable for real-time vision applications because of the time required to obtain the scene sampling and the high computational cost.

Sparse multi-view stereo matching algorithms seek to balance between speed and quality. They have more input views than binocular stereo and thus have a better chance to produce more accurate depth information. On the other hand, sparse multi-view stereo matching is more likely to be achieved in real-time as compared to dense multi-view counterpart and it is a better candidate as a module in real-time vision applications.

5.2 Sparse Multi-view Camera Setup

Two specific types of camera setups are adopted in the proposed sparse multi-view stereo system. One type of setup is a linear camera array with 5 uniformly spaced cameras, and the other is a cross camera set up with four target cameras positioned to the top, bottom, left, and right of the reference camera with the same separation, i.e, baseline. Figure 5.1 shows both setups. They are chosen because 1) Middlebury vision site [26] provides the standard datasets that are captured using these two setups; and 2) it is easy to use these setups to sample real scenes.

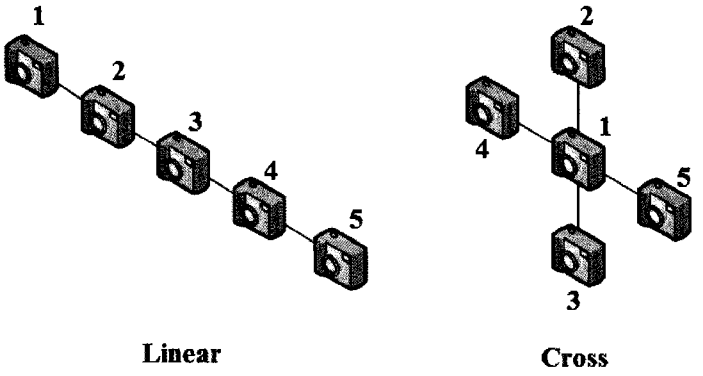


Figure 5.1: Two different camera setups are adopted.

The first linear camera setup is also adopted in other multiple baseline stereo research [32]. The baseline is the distance between the optical centers of two cameras, one of which is referred to as the reference camera and the other the target camera. For a scene point P, its projected pixels onto the reference and target images are separated by disparity d, and its distance with the image plane is denoted as z. The relation between d and z is defined by:

$$d = BF\frac{1}{z} \tag{5.1}$$

where B and F are the baseline and focal length, respectively. Thus, the disparity value is proportional to the baseline length for the same depth value. For example, in the linear camera setup in Figure 5.2, if a scene point is projected to camera 1 at the pixel location (x_1, y_1), to camera 2 at the pixel location (x_2, y_2), and to camera 3 at the pixel location (x_3, y_3), then $(x_1 - x_3) = 2(x_2 - x_3)$ (Note that $y_1 = y_2 = y_3$ as all images are rectified). With camera 3 as the reference camera, if the disparity values to be evaluated against camera 2 is $(1,2,3,\ldots,n)$, then the corresponding disparity values for camera 3 is doubled, *i.e.* $(2,4,6,\ldots,2n)$. This guarantees that the cost volume generated for every reference view has the same depth tessellation pattern and can be later on merged together in the cost merging stage.

In the second cross camera setup, the baselines for every reference-target camera pair have the same length, but with different orientations. It is labelled "Cross" in Figure 5.2. Thus the disparity shift happens in the –x direction in the image plane for camera pair (1,4), in the +x direction for camera pair (1,5), in the -y direction for camera pair (1,2), and in the +y direction for camera pair (1,3). The cost volumes generated with the proper disparity shift direction again has the same depth tessellation pattern and can be later on merged together in the cost merging stage.

5.3 Parzen Window driven Cost Merging

In this section, the detail of the Parzen-window driven cost merging proposed by Vogiatzis *et al.* [46] is discussed.

Given a reference camera and N target cameras, N cost volumes can be generated using any two-frame stereo algorithms. Vogiatzis *et al.* use normalized cross correlation with a square window to compute and aggregate the cost. Let AC_i denote the cost volume calculated using reference camera i, and $AC_i(p, d)$ the

cost of projecting a center view pixel p with depth/disparity d to the reference camera i. The cost merging step needs to combine $AC_i(p,d)$ from all the N reference cameras into a single merged cost $MAC(p,d)$. By assigning each depth level with a matching cost, a new merged cost volume MAC is obtained and ready to be used by a disparity selection algorithm.

The simplest way to do merging is by sum/average, i.e.:

$$MAC(p,d) = [\sum_{i \in N} CV_i(p,d)]/N \tag{5.2}$$

Or a slightly better way proposed in [19] is to use the best half of the target cameras,

$$MAC(p,d) = \sum_{i \in BH_{d,p}(N)} AC_i(p,d)$$

where

$$BH_{d,p}(N) = \{j | AC_j(p,d) \geq median_{k \in N}(AC_k(p,d))\}. \tag{5.3}$$

The aforementioned merging approaches are heuristic and not robust enough against noise, occlusions, or the lack of texture. For some pixels, faulty good matches and noise on the cost-depth curves for all reference cameras can easily disturb the final merged cost-depth curve so that the simple winner-take-all algorithm will fail to assign the correct depth.

Figure 5.2(a) shows all the reversed matching cost curves to be merged, each associated with a reference-target camera pair. Note that the matching cost curves have to be reversed because of the maxima-clustering nature of Parzen window process applied later, thus a bigger value along the reversed matching cost curve indicates a better match. d_0 is the true disparity that some of the curves agree but

others disagree due to occlusion or image noise. The average matching cost curve, which is the dotted curve in (b), apparently misses d_0.

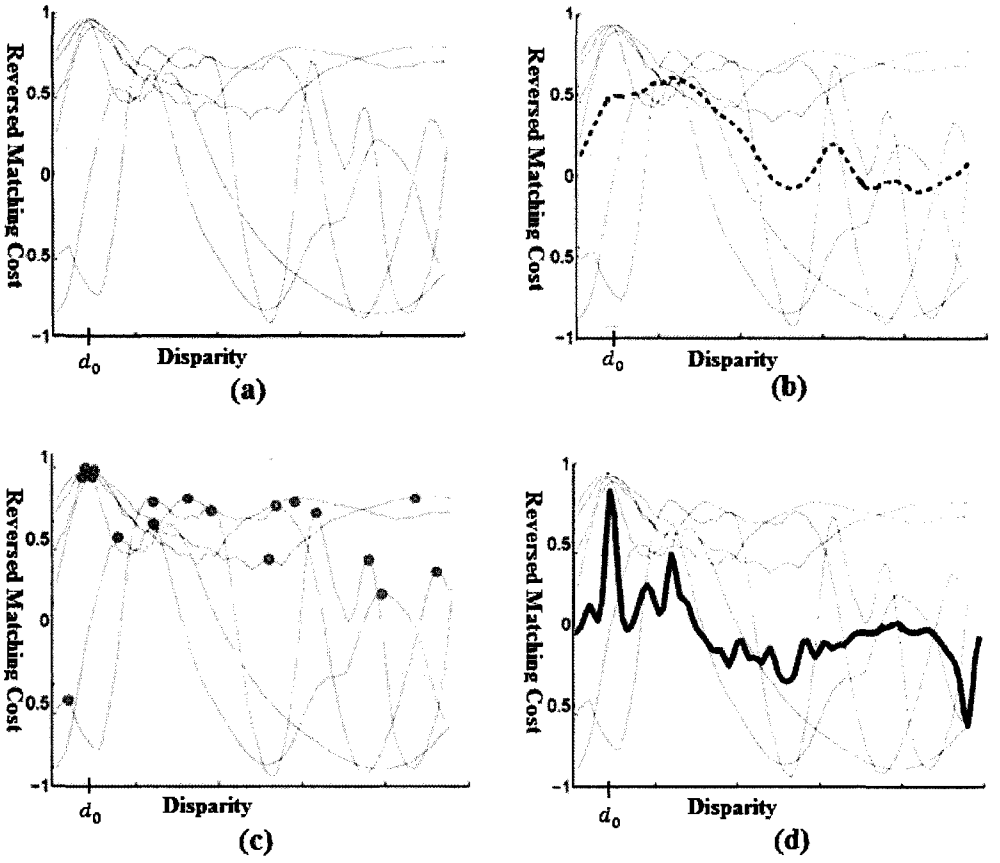


Figure 5.2: How to merge multiple matching cost curves. (Courtesy of George Vogiatzis.)

It is clear that the global maximum of a single cost-depth curve may not correspond to the correct depth. However, at the correct depth, the visible cameras still can give local maximum scores, even if it is not a global one. It is more often than not that more local maxima can be found around the correct depth, while the error due to occlusion can cause only a small number of the curves not to reach their corresponding local maxima. So, detecting all local maxima and finding a

way to reinforce those local maxima that are close to each other can give a more robust and noise-free merged curve.

This reinforcement is achieved by applying the probability model coined by Parzen [33]. The Parzen-window density estimation, or kernel density estimation, is essentially a data interpolation technique. Given some observations of a random variable, Parzen window can reconstruct any data point by aggregating over the known observations with a kernel (Gaussian-kernel is one example). Thus the entire population can be interpolated.

To apply Parzen window in the multiview stereo scenario, the observation data points are not randomly chosen but local maxima d_k in all the reversed matching cost curves, as depicted by the brown dots in Figure 5.2(c). They are detected by:

$$\frac{\delta CV_i}{\delta d}(p, d_k) = 0$$

and

$$\frac{\Delta^2 AC_i}{\delta d^2}(p, d_k) > 0 \tag{5.4}$$

Then the merging is conducted with Parzen window:

$$MCV(p, d) = \sum_{i \in N} \sum_{k} CV_i(p, d_k) \cdot G(d - d_k) \tag{5.5}$$

where G is a Gaussian kernel that makes sure that local optima contribute more to nearby depth candidates. The Parzen window process achieves mutual reinforcement of local maxima as Gaussian kernel applies a bigger weight to a nearby than a farther maximum. If a data point is close to more local maxima, it will have a bigger reconstructed value. Thus, the merging process is robust

against occlusions and image noise. The merged costs are illustrated in 5.2(d) as the solid curve.

5.4 Proposed System

The proposed algorithm performs four steps: cost volume generation, cost volume aggregation, cost volume merging, and disparity selection, as depicted in Figure 5.3.

With the reference view and several target views, the cost volume generation step calculates one cost volume for each reference-target camera pair using any matching cost function. The method to select disparity levels for each reference-target image pair is discussed in the next section. And the matching cost function used here is a simple squared difference of image intensities.

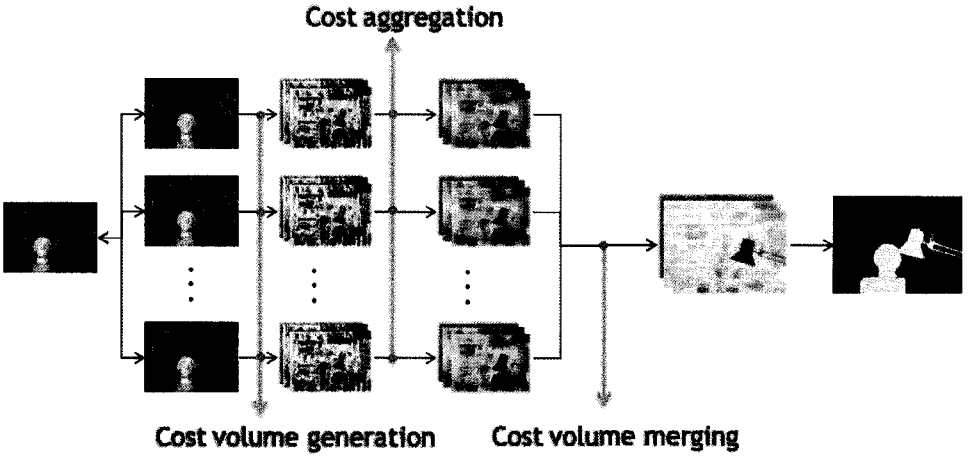


Figure 5.3: System flow of the proposed algorithm.

Then cost aggregation is applied to every single cost volume. The GPU ASW cost aggregation [16] is again used here due to its speed and relative good performance. SSD (sum of squared differences) is also implemented for comparison.

Then the multiple cost volumes are merged into a single cost volume in the second step. A novel way to merge cost volumes combining the occlusion robust photo-consistency metric proposed by Vogiatzis et al. [46] and the ASW idea [52] is presented.

Finally the disparity selection step uses winner-take-all to locally decide the best disparity for every single pixel in the center view. These three parts are seamlessly and independently joined together.

The number of matching cost curves used in the Parzen window approach depends on the number of reference images. With a sparse camera array setup, the number of curves may be too few to guarantee the effectiveness of the Parzen window technique. When one bad pixel sample out of a small number of reference views is present, the disturbance can be big and the Parzen window technique may fail to recover the correct depth.

To minimize the effects from noisy input or occlusion, we can use curves from neighbour pixels. This approach is valid since neighbour pixels usually have similar disparity values and hence their curves can help to reinforce the local optima at the correct depth. It is also more effective because using all pixels within a small 3×3 neighbourhood gives 9 times more curves to work with, which can effectively reduce the effects of noisy input and occlusion.

When using curves from neighbour pixels, we should try to use only pixels that have the same depth as the center pixel. Hence, the question of which neighbouring pixels we should use is similar to the one we face in the cost aggregation step. Therefore, the best cost aggregation technique – ASW – is the perfect solution. With proper weights assigned to the neighbouring pixels within a big square neighbourhood, tens, or even hundreds of curves are made available to merge for a single pixel, and those neighbouring pixels with a better chance of having the same depth value will have higher weights and hence contribute more in the merging process.

5.4.1 Multiple cost volume merging

CPU version.

The formula we use for merging in the CPU version is:

$$MCV(p,d) = \sum_{q \in win(p)} \sum_{i \in N} \sum_{k} CV_i(p,d) \cdot CV_i(q,d_k) \cdot G(d-d_k) \cdot w(p,q) \tag{5.6}$$

where *win(p)* is the neighbourhood of pixel p where adaptive support-weight applies, and *w(p,q)* is the weight calculated using (2.2). d_k is a local maximum from the reversed cost matching curve for q.

Notice that there is a slight difference between the original equation (5.5) and equation (5.6). An extra term $AC_i(p.d)$ is introduced. Consider a pixel-depth combination, where the matching scores for all reference cameras, *i.e.* $AC_i(p.d)$, are low. There is very little chance for the true depth to fall on depth *d*. But if *d* is close to many local optima on the cost-depth curves, there is a possibility that *MAC(p, d)* is merged as the global optimum. These false global optima can be reduced by taking into considerations the actual matching scores at depth *d*. The experimental results show a 2%-20% reduction in error rates in all tested datasets by replacing equation (5.6) with equation (5.5).

The CPU version following equation (5.6) is implemented and tested to fully demonstrate the effectiveness of the proposed technique.

GPU version

In order to achieve real-time performance, we have to simplify the equation to make it more efficient for GPU implementation.

Notice that, if we replace $AC_i(p,d)$ by $AC_i(q,d)$ in equation (5.6), the formulation can be rewritten as:

$$MAC(p,d) = \sum_{q \in win(p)} \sum_{i \in N} \sum_{k} AC_i(q,d) \cdot AC_i(q,d_k) \cdot G(d-d_k) \cdot w(p,q)$$
$$= \sum_{q \in win(p)} w(p,q) \left(\sum_{i \in N} \sum_{k} AC_i(q,d) \cdot AC_i(q,d_k) \cdot G(d-d_k) \right) \tag{5.7}$$

The replacement will not have a big impact to the result since neighbour pixels with high support weight have a good chance of having a similar curve with the pixel of interest. And this modification enables us to separate the computation into 2 passes. The first pass is the Parzen-window cost volume merging without adaptive support-weight. This pass is parallel and is adapted to the GPU implementation easily. The second pass applies the GPU ASW onto the merged cost volume to complete the cost merging process.

5.4.2 Disparity selection

After cost volume merging, *MAC* now holds a single cost volume, to which the winner-take-all optimization can be applied for finding the best disparity value for each pixel p, the same as equation (3.10).That is:

$$Disp(p) = \text{argmin}_d MAC(p,d) \tag{5.8}$$

5.5 Experimental Results

5.5.1 Experimental setup

Five datasets are tested. The *Tsukuba* and *Santa_ Doll* datasets are from the Multi-view Image Database of the University of Tsukuba. The four reference cameras and the center camera follow the cross configuration. The other three datasets are from the Middlebury stereo evaluation website, namely *Teddy, Cones* and *Venus*. For these three datasets, 10 images are taken from a linear camera

setup. Camera 2 and 6 are used for the standard two-frame stereo evaluation, where 2 is the center camera. We use camera 0, 1, 2, 4, 6, with 2 still being the reference camera. In order to achieve the same disparity range in the two-frame setup, sub-pixel disparity is required for camera 0, 1, and 4. By doing this, we can use the Middlebury stereo website to evaluate our results.

Experiments are performed on a machine running Windows XP Professional with 4GB of system memory, with two AMD 2.21GHz dual-core Opteron CPUs, and two NVIDIA 8800 GTS 512MB GTS graphics cards, only one of which is used in the GPU implementation.

The parameters used in the experiments follow the empirical choices suggested in the original papers. In the GPU ASW algorithm, γc=19.6 and γg=40, as adopted in [16]. The adaptive support-weight window size is 33 in the cost aggregation step and 17 in the cost volume merging step. It is noteworthy that the original Parzen-window technique deals with NCC matching score, where a larger score means a better match. Since we use the difference based cost matching and cost aggregation, where a smaller value means a better match, so we reverse the normalized value before the cost volume merging step so that a bigger score indicates a better match.

5.5.2 Disparity results

Figures 5.3-5.6 show the results of running different cost volume merging algorithms on the *Tsukuba*, *Teddy*, *Cones* and *Venus* datasets. Of all the four groups of figures, the top rows show four disparity results from four single reference view cost volumes. Due to the visibility problem, these disparity maps all have errors in the occluded regions. In the bottom rows, the first ones from the left are from the averaging cost volume merging technique; the second ones use the original Parzen-window technique; the third ones are obtained from the proposed technique; and the last ones are the ground truth. Though all three

approaches produce better disparity maps than the two-frame results, the ones generated using the new technique are the most visually appealing.

Table 5.1 gives the statistical comparison of different techniques implemented on CPU. All of the error rates are evaluated by the Middlebury stereo vision website. Averaging cost volume merging and Parzen-window cost volume merging definitely improve upon the two-frame results, since more cameras are used. Parzen-window performs arguably better than averaging, mainly because the sparse camera set-up does not give enough cost-disparity curves for Parzen window to work on. The new technique performs well for most of the datasets. With more cameras, it easily outperforms all two-frame local stereo algorithms, but is still behind some of the best global optimization methods.

A special note on the *Venus* dataset is that, due to the rich texture and the use of simple sub-pixel linear interpolation, the cost volumes generated with camera 0, 1, 4 are very poor, producing around 10% error rate. Thus the effectiveness of the merging techniques is not significant. As well, there is a sharp drop of disparity result quality for average merging and simple Parzen window merging. We plan to investigate this problem further in future research.

5.5.3 With poor cost volume inputs

When the aggregated costs are generated using the sum-of-squared-differences, instead of the simplified adaptive support-weight approach, the cost volumes (AC_i) can be of poor quality. Here we use a 5×5 SSD window to generate the cost volumes for the *Tsukuba* and *Santa_Doll* datasets, and then apply different cost volume merging techniques for comparison.

The results, shown in Figure 5.7, demonstrate that even with poor cost volume inputs, our technique is still robust enough to obtain acceptable results. From the left, the first, middle and right are obtained respectively using the averaging cost

volume merging technique, the original merging technique, and our proposed method.

5.5.4 GPU version results

The GPU version provides real-time performance. Indeed, it runs at around 70 FPS for the *Tsukuba* dataset, 45 FPS for the *Venus* dataset and 15 FPS for the *Cones* and the *Teddy* datasets on one NVIDIA 8800 GTS 512MB graphics card. The disparity results are shown in Figure 5.8. Despite some loss in accuracy because of the simplified formula and GPU tailored implementation, the disparity results are still quite acceptable and can definitely be used as a module in real-time vision applications.

	nocc	all	disc	nocc	all	Disc
	TSKUKUBA			*TEDDY*		
Cen-right	2.27	3.61	11.20	12.0	19.5	23.8
Avg	1.56	1.85	8.37	9.42	11.7	18.8
Parzen	1.41	1.76	7.57	11.1	12.8	19.1
Proposed	1.08 *5*	1.39 *3*	5.79 *6*	7.74 *17*	9.95 *5*	17.1 *16*
	CONES			*VENUS*		
Cen-right	12.6	18.9	19.4	3.45	4.28	16.7
Avg	5.76	7.94	12.5	4.59	5.26	13.8
Parzen	5.83	8.00	12.6	8.90	9.42	18.7
Proposed	3.05 *6*	4.31 *1*	7.48 *2*	2.30 *30*	2.90 *27*	10.7 *25*

Table 5.1: Error rates evaluated from the Middlebury website, with the Tsukuba, Teddy, Cones and Venus datasets. The Cen-right rows hold results for the two-frame stereo with center and right camera. The Avg rows show results from averaging all cost volumes. Parzen denotes the original Parzen cost volume merging technique. The results in the rows labelled Proposed are from our proposed technique, and the numbers in italic are ranks from the Middlebury stereo evaluation website.

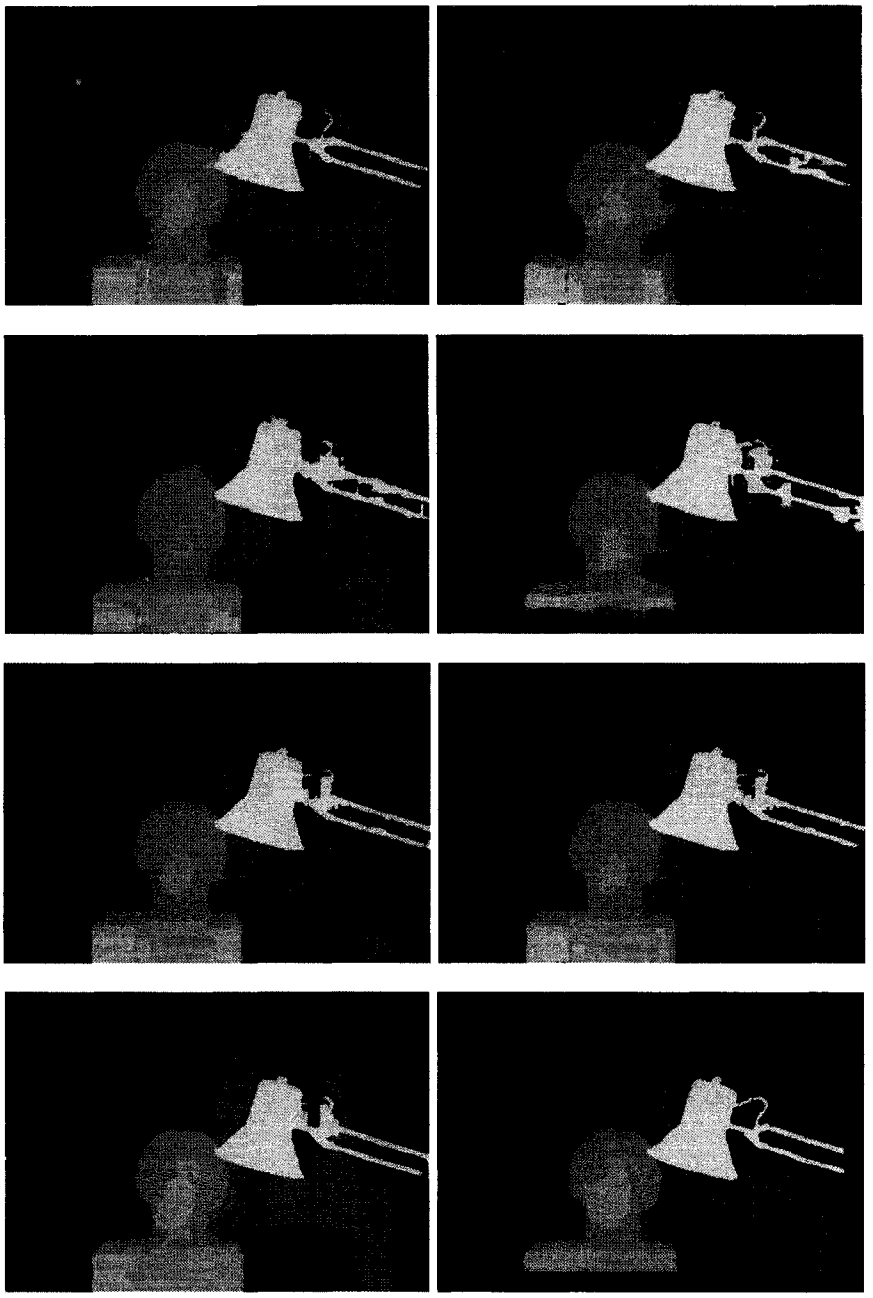


Figure 5.4: Results for the Tsukuba dataset. The top two rows show disparity results for individual target views; the third row, in the order from left to right, shows the result using average merging and Parzen-window merging; the last row has result from the proposed method, and the ground truth. The same layout applies to Figure 5.4, Figure 5.5 and Figure 5.6.

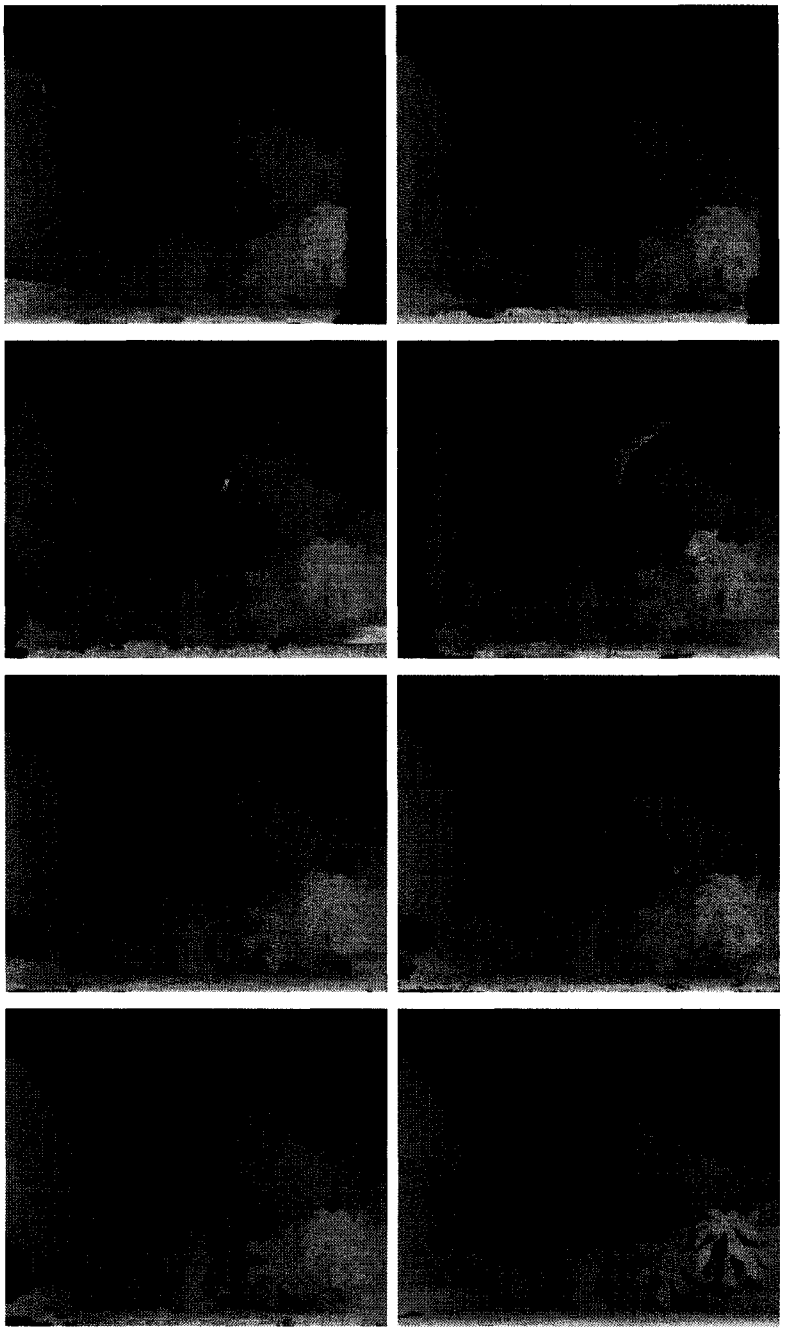


Figure 5.5: Results for the Teddy dataset

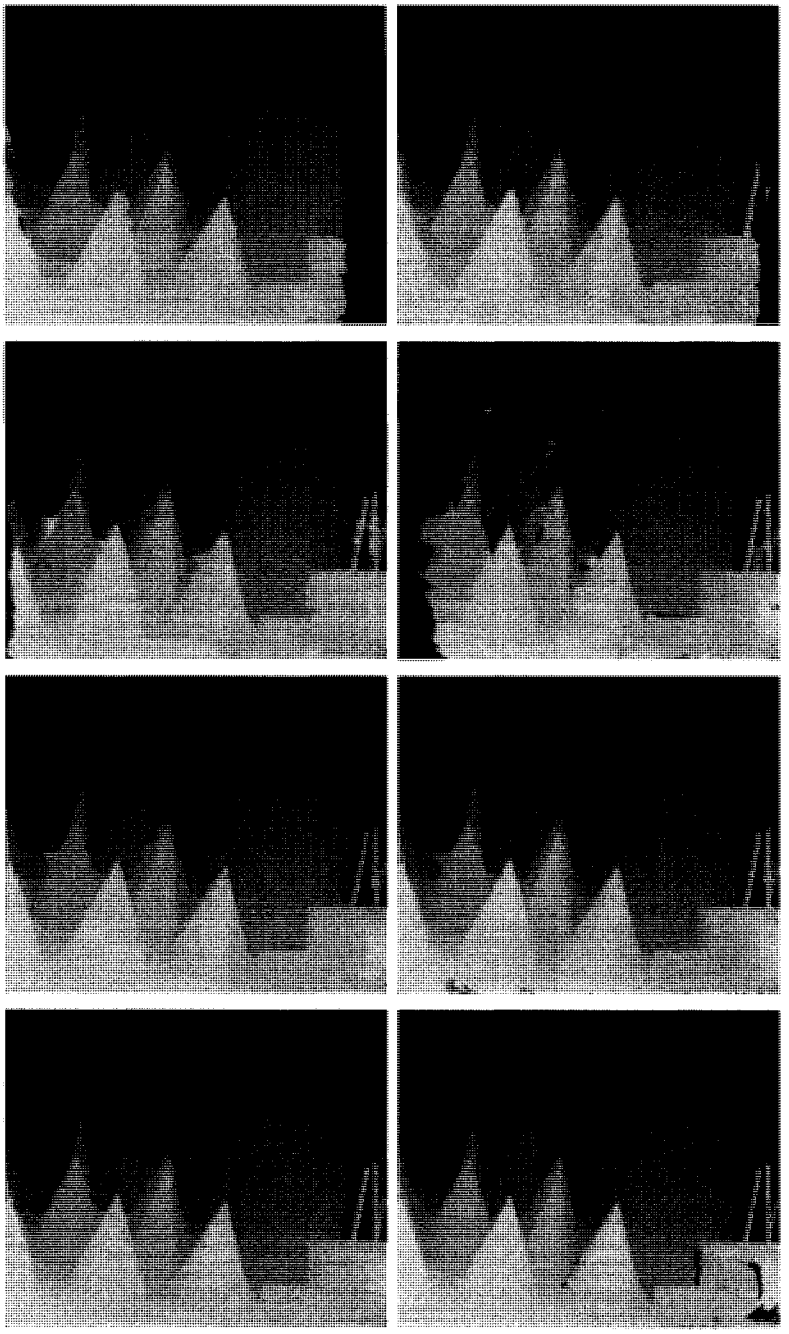


Figure 5.6: Results for the Cones dataset.

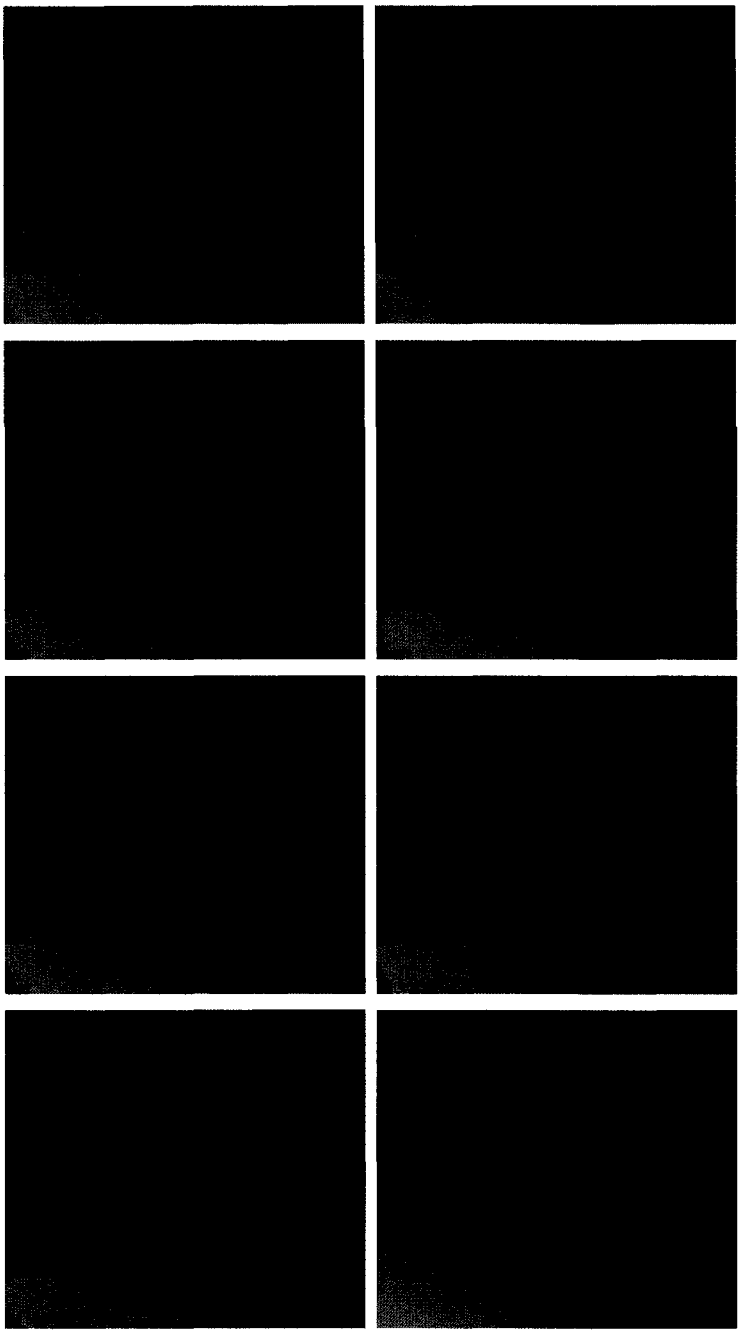


Figure 5.7: Results from the Venus dataset.

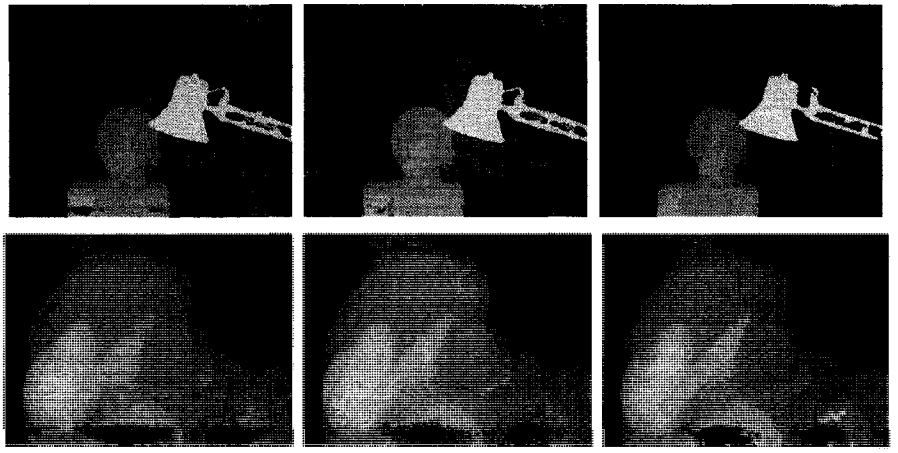


Figure 5.8: Results for the Tsukuba and Santa_Claus dataset with SSD generated cost volumes.

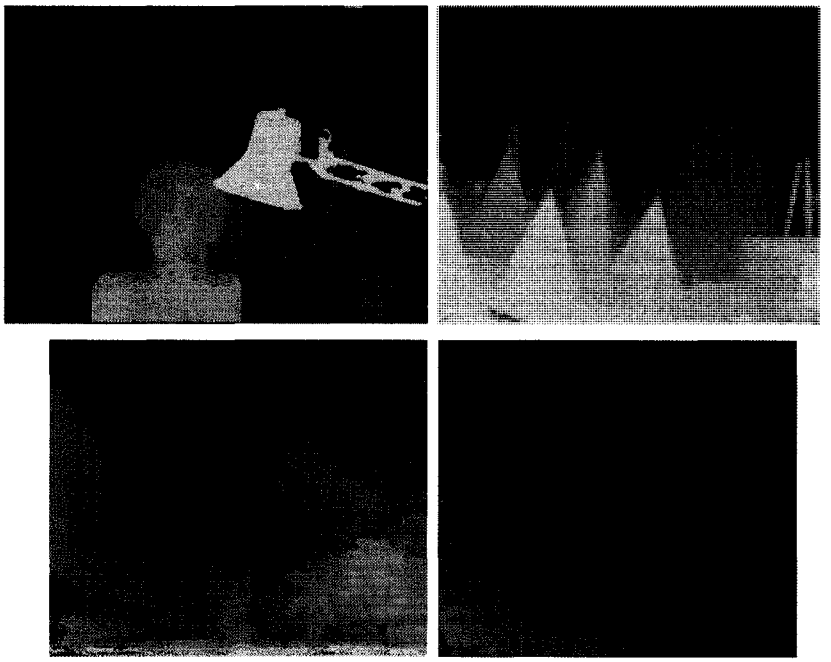


Figure 5.9: Results of using the GPU implementation.

5.6 Summary

A new local multi-view stereo algorithm is presented in this chapter. The algorithm is designed for highly parallel systems and can achieve real-time performance. The adaptive support-weight cost aggregation idea and Parzen-window cost volume merging idea are combined to achieve a robust cost volume merging module. Our encouraging results show the effectiveness of the proposed method. A simplified real-time GPU version is implemented as well.

Chapter 6:
Conclusion and Future Work

To extend the usefulness of the ASW cost aggregation method, three new algorithms are developed in this thesis.

A new local stereo matching algorithm is presented in Chapter 3, which combines adaptive-weight cost aggregation with slanted surface modeling. In order to achieve superior disparity result, the plane fitting idea is used to extend aggregation into the 3rd dimension and expands through multiple disparity levels. Such an extension is essential to better model slanted surfaces. The results demonstrate the effectiveness of the new algorithm. Some extension works can be done along this direction. First, the raw disparity map used to generate DPO has a direct impact on the accuracy of extracted slanted surfaces and later can influence the disparity result. In fact, the experimental results presented in this thesis show that DPO generated from ground truth improves the final result by 10% to 50% on four datasets. Hence, a better DPO generation approach may improve the input disparity result. Another possible avenue to pursue is to get a better initial estimate by incorporating range images [51]. Along the direction of using GPU, the parallel characteristic of this algorithm can be fully exploited by porting to GPU.

Chapter 4 presents the endeavour of using CUDA to fully implement the ASW algorithm onto GPU without much loss of quality. The quadtree based approach replaces pixel primitives in stereo matching by square-shaped segments, reduces

the number of primitives to be processed, and improves the inner-segment smoothness by assigning uniform disparity values for every segment. Although challenges still remain, improvements in both quality and speed are still feasible. First of all, the cue of Euclidean distance in segment-to-segment weight is approximated by the distance of their segment centers. This reduces the credibility of the weights. Also, square-shaped segments, although efficient to process in GPU, are quite restrictive and render the segmentation quality vulnerable to high-textured areas and image noise as a single heterogeneous pixel in a large homogenous area, which will lead to more segments than desired. A more general segmentation method which allows non-rectangular shaped segments may be more effective. Thirdly, the neighbourhood to be aggregated is still of a fixed size, which may not adapt to low-textured areas and high-textured areas well. Using a fixed number of segments that bear the biggest weights within the neighbourhood of the segment-of-interest is one possible solution. A better structure has to be used in order to find the nearest neighbours in weights rather than just in spatial distance. Using a KD-tree is one option, and a CUDA based real time photon mapping application has already been proposed by Zhou *et al.* [55], which features efficient construction of KD-tree of photons and KNN searches of any photon.

The new multi-view stereo algorithm based on Parzen-window and ASW discussed in Chapter 5 shows promising results and robustness against occlusions. One possible avenue of improvement is to refine the cost volume results for every view by merging other views using the new method, and then iteratively update each cost volume to get even better results. Also, a complete system that can process multiview images in real-time to produce high quality disparity results is worthy to be developed to fully demonstrate the effectiveness of the new methods presented in this thesis..

Finally, the three algorithms discussed in this thesis try to enhance local stereo matching from three perspectives, namely subpixel disparity accuracy, real-time

speed under ASW aggregation, and occlusion handling. How to integrate the presented algorithms to achieve all three objectives will be worth investigating.

Bibliography

[1] Barnard, S.; Fishler, M. "Computational stereo." *ACM Computing Surverys,* vol.14, pp. 553-572, 1982.

[2] Bobick, A. F.; Intille, S. "Large occlusion stereo." *International Journal of Computer Vision,* vol.33, pp.181-200, 1999.

[3] Brown, M. Z.; Burschka, D.; Hager, G. D. "Adcances in computational stereo." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol.25, pp.993-1008, 2003.

[4] Buck, I.; Foley, T.; Hron, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. "For GPUs: stream computing on graphics hardware." *ACM Transactions on Graphics*, vol.23, pp.777-786, 2004.

[5] Chan, S.; Wong, Y.-P.; Daniel, J. "Dense stereo correspondence based on recursive adaptive size muti-windowing." *Proceedings of Image and Vision Computing New Zealand.* Pp.256-259, 2003.

[6] Criminisi, A.; Blake, A.; Rother, C. "Efficient dense stereo with occlusions for new view-synthesis by four-state dynamic programming." *International Journal of Computer Vision,* vol.71, pp.89-110, 2007.

[7] Criminisi, A.; Shotton, J.; Blake, A.; Torr, P. H. S. "Gaze manipulation for one-to-one teleconferencing." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.191-198, 2003.

[8] CUDPP: CUDA Data Parallel Primitives Library. *http://www.gpgpu.org/developer /cudpp*

[9] Dhond, U. R.; Aggarwal, J. K. "Structure from stereo - a review." *IEEE Transactions on Systems, Man and Cybernetics,* vol.19, pp.1489-1510, 1989.

[10] Drouin, M. A.; Trudeau, M.; Roy, S. "Fast multi-baseline stereo with occlusion." *Proceedings of International Confererence on 3-D Digital Imaging and Modeling,* pp.540-547, 2005.

[11] Drouin, M. A.; Trudeau, M.; Roy, S. "Geo-consistency for wide multi-camera steero." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.885-891, 2005.

[12] Farrugia, J.-P.; Horain, P.; Guehenneux, E.; Alusse, Y. "GPUCV: A Framework for Image Processing Acceleration with Graphics Processors." *IEEE Conference on Multimedia and Expo,* pp.585-588, 2006.

[13] Forstmann, S.; Kanou, Y.; Ohya, J.; Thuering, S.; Schmitt, A. "Real-time stereo by using dynamic programming." *Computer Vision and Pattern Recognition Workshop,* p.29, 2004.

[14] Fusiello, A.; Roberto, V. ; E. Trucco. "Efficient stereo with multiple windowing." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.858-863, 1997.

[15] Geiger, D.; Ladendorf, B.; Yuille, A. "Occlusions and binocular stereo." *International Journal of Computer Vision*, vol.14. pp.211-226, 1995.

[16] Gong, M.-L.; Yang, R.-G.; Wang, L.; Gong, M.-W. "A performance study on different cost aggregation approaches used in real-time stereo

matching." *International Journal of Computer Vision*, vol.75, pp.283-296, 2007.

[17] Gong, M.-L.; Yang, Y.-H. "Image-gradient guided real-time stereo on graphics hardware." *Proceedings of International Conference on 3-D Digital Imaging and Modeling,* pp.548-555, 2005.

[18] Kanada, T.; Okutomi, M. "A stereo matching algorithm with an adaptive window: theory and experiment." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* pp.920-932, 1994.

[19] Kang, S.-B.; Szeliski, R.; Chai, J. "Handling occlusions in dense multi-view stereo." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pp.103-110, 2001.

[20] Klaus, A.; Sormann, M.; Karner, K. "Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure." *Proceedings of International Conference on Pattern Recognition,* pp.15-18, 2006.

[21] Kolmogorov, V.; Zabih, R. "Computing visual correspondence with occlusions via graph cuts." *Proceedings of IEEE International Conference on Computer Vision,* pp.508-515, 2001.

[22] Kutulakos, K.; Seitz, S. M. "A theory of shape by space carving." *International Journal of Computer Vision,* vol.38, pp.199-218, 2000.

[23] Marr, D.; Nishihara, H. K. "Representation and recognition of the spatial organization of three-dimensional shapes." *Proceedings of the Royal Society of London*, vol.200, pp.269-294, 1978.

[24] Marr, D.; Poggio, T. "A computational theory of human stereo vision." *Proceedings of the Royal Society of London,* vol.204, pp.301-328, 1979.

[25] McCool, M.; Qin, Z.; Popa, S. T. "Shader Meteprogramming." *Proceedings of SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop,* pp.57-68, 2002.

[26] Middlebury Stereo Vision Page. *http://vision.middlebury.edu/stereo/*

[27] Nakamura, Y.; Matsuura, T.; Satoh, K.; Ohta, Y. "Occlusion detectable stereo - occlusion patterns in camera matrix." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.371-378, 1996.

[28] Nelson, D.; Yang, Y.-H. "Evaluation of Constructable Match Cost Measures for Stereo Correspondence Using Cluster Ranking." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* 2008.

[29] NVIDIA CUDA Occupancy Calculator. *http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html*

[30] NVIDIA CUDA Programming Guide. *http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf*

[31] NVIDIA CUDA Zone. *http://www.nvidia.com/object/cuda_home.html*

[32] Okutomi, M.; Kanade, T. "A multiple baseline stereo." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol.15, pp.371-348, 1993.

[33] Parzen, E. "On estimation of a probability density function and the mode." *Annals of Mathematical Statistics,* vol.33, pp.1065-1076, 1962.

[34] Scharstein, D.; Szeliski, R. "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms." *International Journal of Computer Vision,* vol.47, pp.7-42, 2002.

[35] Seitz, S. M.; Curless, B.; Diebel, J.; Scharstein, D.; Szeliski, R. "A comparison and evaluation of multi-view stereo reconstruction algorithms." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.519-528, 2006.

[36] Seitz, S. M.; Dyer, R. C. "Photorealistic scene reconstruction by voxel coloring." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* vol.1067-1073, 1997.

[37] Seltzer, J. *Desktop image-based rendering.* Master Thesis. University of Alberta Library, 2006.

[38] Sengupta, S.; Harris, M.; Zhang, Y.; Owens, J. D. "Scan primitives for GPU computing." *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Grpahics Hardware,* pp.97-106, 2007.

[39] Shen, G.-B.; Gao, G.-P.; Li, S.-P.; Shum, H.-Y.; Zhang, Y.-Q. "Accelerated video decoding with generic GPU." *IEEE Transactions on Circuits and Systems for Video Technology,* vol.15, pp.685-693, 2005.

[40] Sinha, S.; Pollefeys, M. "Multi-view reconstruction using photo-consistency and exact silhouette constraints: a maximum-flow formulation." *Proceedings of IEEE International Conference on Computer Vision,* pp.349-356, 2005.

[41] Sun, J.; Li, Y.; Kang, S.-B.; Shum, H.-Y. "Symmetric stereo matching for occlusion handling." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* vol.399-406, 2005.

[42] Tao, H.; Sawhney, H. S. "Global matching criterion and color segmentation based stereo." *Proceedings of IEEE Workshop on Applications of Computer Vision,* pp.246-253, 2000.

[43] Tao, H.; Sawhney, H. S.; Kumar, R. "A global matching for stereo computation." *Proceedings of IEEE International Conference on Computer Vision,* pp.532-539, 2001.

[44] Tombari, F.; Mattoccia, S.; Di Stefano, L. "Segmentation-based adaptive support for accurate stereo correspondence." *Proceedings of Pacific-Rim Symposium on Image and Video Technology,* pp.427-438, 2007.

[45] Tombari, F.; Mattoccia, S.; Di Stefano, L.; Addimanda, E. "Classification and evaluation of cost aggregation methods for stereo correspondence." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* 2008.

[46] Vogiatzis, G.; Esteban, C. H.; Torr, P. H. S.; Cipolla, R. "Multi-view stereo via volumetric graph-cuts and occlusion robust photo-consistency." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol.29, pp.2241-2246, 2007.

[47] Wang, L.; Gong, M.-W.; Gong, M.-L.; Yang, R.-G. "How far can we go with local optimization in real-time stereo matching." *Proceedings of International Symposium on 3D Data Processing, Visualization, and Transmission,* pp.129-136, 2006.

[48] Wang, L.; Liao, M.; Gong, M.-L.; Nister, D. "High quality real-time stereo using adaptive cost aggregation and dynamic programming." *Proceedings of International Symposium on 3D Data Processing, Visualization, and Transmission,* pp.798-805, 2006.

[49] Xu, Y.; Wang, D.; Feng, T.; Shum, H.-Y. "Stereo computation using radial adaptive windows." *Proceedings of International Conference on Pattern Recognition,* pp.595-598, 2002.

[50] Yang, Q.; Wang, L.; Yang, R.-G.; Nister, D. "Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.2347-2354, 2006.

[51] Yang, Q.-X.; Yang, R.-G.; Davis, J.; Nister, D. "Spatial-depth super resolution for range images." *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition,* pp.1-8, 2007.

[52] Yoon, K.-J.; Kweon, I.-S. "Adaptive support-weight approach for correspondence search." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol.28, pp.650-656, 2005.

[53] Zhang, Y.-L.; Gong, M.-L.; Yang, Y.-H. "Local stereo matching with 3D adaptive cost aggregation for slanted surface and sub-pixel accuracy." *Proceedings of International Conference on Pattern Recognition,* 2008.

[54] Zhang, Y.-L.; Gong, M.-L.; Yang, Y.-H. "Real-time multi-view stereo using adaptive-weight parzen-window and local winner-take-all optimization." *Proceedings of Fifth Canadian Conference on Computer and Robot Vision,* pp.113-120, 2008.

[55] Zhou, K.; Hou, Q.-M.; Wang, R.; Guo, B.-N. Real-time KD-tree construction on graphics hardware. Technical Report, Microsoft Research Asia, 2008.

Appendix I:

The pseudo code for direct porting ASW onto GPU runs like this:

Algorithm AI.1

```
procedure AdaptiveSupportWeightCostAggregation
  in:  float cv[][], int dispRange, int szAggrWin, texture targetIm
  out: float cv[][]
begin
for i = 0 to dispRange-1
  for each pixel position in parallel
    aggrCost ← 0
    po ← the colour value for the pixel position from targetIm
    for each neighbour pixel in the aggregation window restricted by szAggrWin
      pn ← the colour value for the neighbour pixel from targetIm
      wt ← calculate the weight using po, pn, and neighbour pixel offset
      cost ← the matching cost for neighbour pixel position from cv[i]
      aggrCost ← aggrCost + cost*wt
    end
    update cv at with aggrCost at the corresponding pixel position and disparity i
end
```

Note that in the code, *cv* is a two-dimensional array. Thus *cv*[i] means a one-dimensional array of size $imageWidth \times imageHeight$ that stores all the costs for disparity level *i*. *dispRange* is the number of disparity levels for the dataset. *targetIm* is the texture reference that stores the target image. The reference image is not needed here because the weight calculation uses the pixel-of-interest and its neighbour pixels in the target image. *szAggrWin* denotes the size of the local aggregation window. *wt* is calculated using equation (AI.1). Code fraction

described as **in parallel** runs as CUDA kernels on GPU. The most inner for loop performs the aggregation described in equation (AI.2).

$$w(u, v, m, n) = \exp\left(-\left(\frac{\Delta c_{u,v,m,n}}{\gamma_c} + \frac{\Delta g_{u,v,m,n}}{\gamma_g}\right)\right) \tag{AI.1}$$

$$AC(u, v, d) = \frac{\sum_{m,n\in[-r,r]} \left(\begin{matrix} w(u, v, m, n) \\ \cdot C(u + m, v + n, d) \end{matrix} \right)}{\sum_{m,n\in[-r,r]} w(u, v, m, n)} \tag{AI.2}$$

Appendix II:

Pseudo codes for two improved ways of direct poring ASW to GPU are listed below:

Algorithm AII.1

```
procedure AdaptiveSupportWeightCostAggregationImproved1
  in:  float cv[][], int dispRange, int szAggrWin, texture targetIm
  out: float cv[][]
begin
  for each pixel position in parallel
    aggrCost[dispRange] ← 0, and aggrCost is allocated on shared memory
    po ← the colour value for the pixel-of-interest from targetIm
    for each neighbour pixel in the aggregation window restricted by szAggrWin
      pn ← the colour value for the neighbour pixel from targetIm
      wt ← calculate the weight using po, pn, and neighbour pixel offset
      for i = 0 to dispRange-1
        cost ← the matching cost for neighbour pixel position from cv[i]
        aggrCost[i] ← aggrCost + cost*wt
    end
    update cv at with aggrCost at the corresponding pixel position
end
```

Algorithm AII.2

```
procedure AdaptiveSupportWeightCostAggregationImproved2
  in:  float cv[][], int dispRange, int szAggrWin, texture targetIm
  out: float cv[][]
begin
  for each pixel position in parallel
    allocate targetImPatch[] on shared memory, with proper size
```

```
    allocate cvPatch[] on shared memory, with proper size
    read in portion of targetImPatch[] from targetIm
    read in portion of cvPatch[] from cv
    synchronize()
    po ← the colour value for the pixel-of-interest from targetImPatch
    for each neighbour pixel in the aggregation window restricted by szAggrWin
      pn ← the colour value for the neighbour pixel from targetImPatch
      wt ← calculate the weight using po, pn, and neighbour pixel offset
      for i = 0 to dispRange-1
        cost ← the matching cost for neighbour pixel position from cvPatch
        aggrCost[i] ← aggrCost + cost*wt
    end
    update cv with aggrCost at the corresponding pixel position
end
```

Algorithm AII.1 shows how to process all disparity levels in one kernel; and AII.2 how to cooperate between kernels to achieve better memory access scheme. Note that *targetImPatch* and *cvPatch* are the supersets of all neighbourhoods for every pixel in the block. synchronize () is necessary to make sure all kernels in the block have finished initializing their share of the shared memory, as no dirty region in the shared memory is acceptable. The size of *targetImPatch/cvPatch* equals $(2W + K) \times (2W + K)$ under the setup of a $(2W + 1) \times (2W + 1)$ aggregation window size and a $K \times K$ kernel block size, as described in Section 4.2.2.