Belief Propagation Implementation Using CUDA on an NVIDIA GTX 280

Yanyan Xu[1], Hui Chen[1], Reinhard Klette[2], Jiaju Liu[1], and Tobi Vaudrey[2]

[1] School of Information Science and Engineering, Shandong University, China
[2] The *.enpeda..* Project, The University of Auckland, New Zealand

Abstract. Disparity map generation is a significant component of vision-based driver assistance systems. This paper describes an efficient implementation of a belief propagation algorithm on a graphics card (GPU) using CUDA (Compute Uniform Device Architecture) that can be used to speed up stereo image processing by between 30 and 250 times. For evaluation purposes, different kinds of images have been used: reference images from the Middlebury stereo website, and real-world stereo sequences, self-recorded with the research vehicle of the *.enpeda..* project at The University of Auckland. This paper provides implementation details, primarily concerned with the inequality constraints, involving the threads and shared memory, required for efficient programming on a GPU.

1 Introduction

The generation of accurate disparity maps for pairs of stereo images is a well-studied subject in computer vision, and is also a major subject in vision-based driver assistance systems (DAS). Within the *.enpeda..* project, stereo analysis is used to ensure a proper understanding of distances to potential obstacles (e.g., other cars, people, or road barriers). Recent advances in stereo algorithms involve the use of Markov random field (MRF) models; however, this leads to NP-hard energy minimization problems. Using graph cut (GC) or belief propagation (BP) techniques allows us to generate approximate solutions with reasonable computational costs [5].

Implementations of (potentially) global methods such as GC or BP often generate disparity maps that are closer to (if available) the ground truth than implementations of local methods (e.g., correlation-based algorithms). Obviously, global methods take more time for generating the stereo results [13]. Ideally, one wants to combine the accuracy achieved via global methods with the running time of local methods. One option toward achieving this goal is to speed up, for example, a BP implementation without losing accuracy, by taking advantage of the high performance capabilities of Graphic Processing Units (GPUs); available on most personal computing platforms today.

This report describes a General Purpose GPU (GPGPU) implementation of a BP algorithm using the NVIDIA Compute Uniform Device Architecture (CUDA) language environment [1]. The contemporary graphics processor unit (GPU) has huge computation power and can be very efficient for performing data-parallel tasks [7]. GPUs have recently also been used for many non-graphical applications [8] such as in Computer Vision. OpenVIDIA [6] is an open source package that implements different computer vision algorithms on GPUs using OpenGL and Cg. Sinha *et al* [14] implemented a feature based tracker on the GPU. Recently, Vineet [15] implemented a fast graph cut algorithm on the GPU. The GPU, however, follows a difficult programming model that applies a traditional graphics pipeline. This makes it difficult to implement general graph algorithms on a GPU.

[9] reports about BP on CUDA. However, that paper does not mention any details about their implementation of BP on CUDA. This paper explains the implementation details clearly. We then go on to detail important pre-processing steps, namely Sobel edge operator (as performed in [10]) and residual images (as performed in [16]), that can be done to improve results on real-world data (with real-world noise).

This paper is structured as follows. Section 2 specifies the used processors and test data. Section 3 describes the CUDA implementation of the BP algorithm. Section 4 presents the experimental results. Some concluding remarks and directions for future work are given in Section 5.

2 Processors and Test Data

For comparison, in our tests we use a normal PC (Intel Core 2 Duo CPU running at 2.13 GHz and 3 GB memory) or a GPU nVidia GTX 280. GPUs are rapidly advancing from being specialized fixed-function modules to highly programmable and parallel computing devices. With the introduction of the Compute Unified Device Architecture (CUDA), GPUs are no longer exclusively programmed using graphics APIs. In CUDA, a GPU can be exposed to the programmer as a set of general-purpose shared-memory Single Instruction Multiple Data (SIMD) multi-core processors, which have been studied since the early 1980*s*; see [11]. The number of threads, that can be executed in parallel on such devices, is currently in the order of hundreds and is expected to multiply soon. Many applications that are not yet able to achieve satisfactory performance on CPUs may have benefit from the massive parallelism provided by such devices.

2.1 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture, with a new parallel programming model and instruction set architecture. It leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient (parallel) way than on a CPU [12]. CUDA comes with a software environment that allows developers to use C as a high-level programming language. A complier named NVCC generates executable code for GPUs.

CUDA Hardware Model. At the hardware level, the GPU is a collection of multiprocessors (MPs). A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory, see left of Figure 1 [12]. When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block

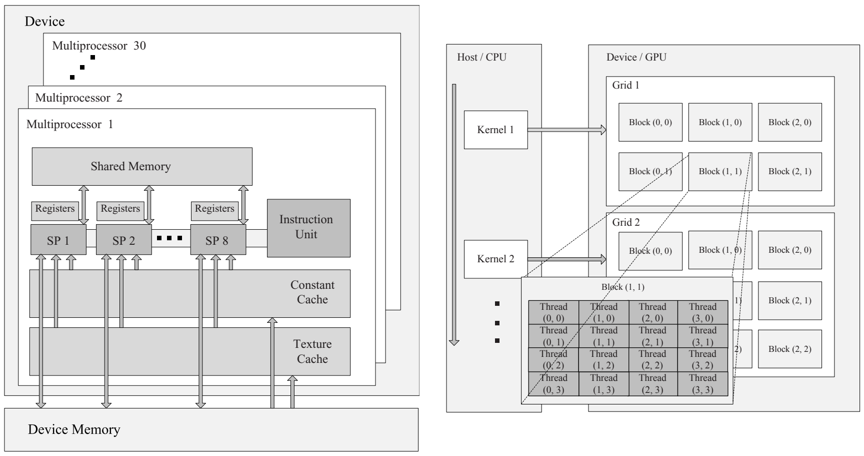

Fig. 1. NVIDIA GeForce GTX 280 CUDA hardware (left) and programming (right) models

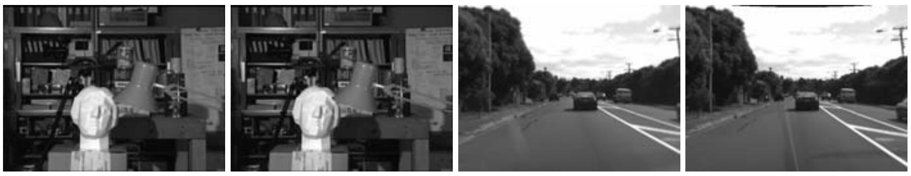

Fig. 2. Test data. Left: Tsukuba stereo pair [13]. Right: real image pair captured by HAKA1.

execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. SP cores in a multiprocessor execute the same instruction at a given cycle. Each SP can operate on its own data, which identifies each multiprocessor to be of SIMD architecture. Communication between multiprocessors is only through the device memory, which is available to all the SP cores of all the multiprocessors. The SP cores of a multiprocessor can synchronize with one another, but there is no direct synchronization mechanism between the multiprocessors. The GTX 280, used in this paper, has 30 MPs, i.e., 240 SPs. The NVIDIA GeForce GTX 280 graphics card has 1 GB of device memory, and each MP has 16 KB of shared memory (shared amongst all SPs).

CUDA Programming Model. To a programmer, parallel portions of an application are executed on the device as *kernels*. One kernel is executed at a time, with many *thread blocks* in each kernel, which is called a *grid* (see right of Figure 1). A thread block is a batch of threads running in parallel and can cooperate with each other by sharing data through shared memory and synchronizing their execution [12]. For the GTX 280, the maximum number $T_{\max}$ of threads equals 512. A *warp* is a collection of threads that are scheduled for execution simultaneously on a multiprocessor. The warp size is fixed for a specific GPU. If the number of threads, that will be executed, is more than the warp size, they are time-shared internally on the multiprocessor. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Using the thread and block IDs, each thread can perform the kernel task on different data. Since the device memory is available to all the threads, it can access any memory location [15].

2.2 Used Test Data

To evaluate our accelerated BP algorithm, different kinds of images have been used: a stereo pair of Tsukuba from the Middlebury Stereo website [13], and real-world stereo sequences, which are captured with HAKA1 (Figure 2), a research vehicle of the *.enpeda..* project at The University of Auckland [4].

3 Implementation of Belief Propagation

Our GPU implementation is based on the "multiscale" BP algorithm presented by Felzenszwalb and Huttenlocher [5]. If run on the original stereo images, it produces a promising result on high-contrast images such as *Tsukuba*, but the effect is not very satisfying for real-world stereo pairs; [10] shows a way (i.e., Sobel preprocessing) how to improve in the latter case, and [16] provides a general study (i.e., for the use of residual input images) for improving results in correspondence algorithms.

3.1 Belief Propagation Algorithm

Solving the stereo analysis problem is basically achieved by pixel labeling: The input is a set P of pixels (of an image) and a set L of labels. We need to find a labeling $f : P \rightarrow L$ (possibly only for a subset of P). Labels are, or correspond to disparities which we want to calculate at pixel positions. It is general assumption that labels should vary only smoothly within an image, except at some region borders. A standard form of an energy function, used for characterizing the labeling function f, is (see [2]) as follows:

$$E(f) = \sum_{p \in P} \left(D_p(f_p) + \sum_{(p,q) \in A} V(f_p - f_q) \right) \tag{1}$$

$D_p(f_p)$ is the cost of assigning label f_p to pixel $p \in \Omega$, where Ω is an M (rows) $\times N$ (columns) pixel grid; the discontinuity term $V(f_p - f_q)$ is the cost of assigning labels f_p and f_q to adjacent pixels p and q. Full details of this equation are found in [5].

As a global stereo algorithm, BP always produces good results (in relation to input data !) when generating disparity images, but also has a higher computational time than local stereo methods. Sometimes we require many iterations to ensure convergence of the message values, and each iteration takes $O(n^2)$ running time to generate each message where n corresponds to the number of possible disparity values (labels). In [5], Felzenszwalb and Huttenlocher present the following methods to speed up the BP calculations.

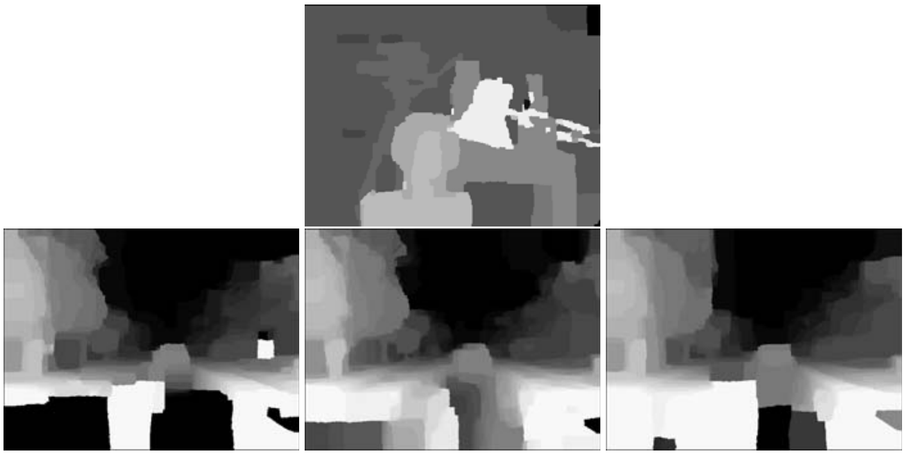

Fig. 3. Example output on test data (see Figure 2 for original images). Top: disparity (colour encoding: dark = far, light = close) for Tsukuba. Bottom: disparity results for a HAKA1 sequence (left to right): on original pair, after applying a Sobel operator, and after mean residual processing.

First, a *red-black method* is provided. Pixels are divided in being either *black* or *red*, at iteration t, messages are sent from black pixels to adjacent red pixels; based on received messages, red pixels sent at iteration $t + 1$ messages to black pixels, and thus the message passing scheme adopts a red-black method, which allows us that only half of all messages are updated at a time.

The *coarse-to-fine algorithm* provides a second method for speeding up BP, and is useful to achieve more reliable results. In the coarse-to-fine method, a Gaussian pyramid is used having L levels (where $L = 0$ is the original image size). Using such a pyramid, long distances between pixels are shortened, this makes message propagation more efficient. This increases the reliability of calculated disparities and reduces computation time without decreasing the disparity search range.

3.2 Belief Propagation on CUDA

BP algorithms have been implemented on the GPU in the past several years: [3] and [17] describe GPU implementations of BP on a set of stereo images. However, each of these implementations uses a graphics API rather than CUDA. Grauer-Gray [9] had implemented BP using CUDA, but did not discuss possible improvements for real-world stereo pairs which always accompany various types of noise, such as different illumination in left and right image, which causes BP to fail.

In our CUDA BP implementation, we define four kernel functions on the GPU, plus the mandatory memory allocation, loading images to GPU global memory, and retrieving the disparity image from the GPU global memory.

1. Allocate GPU global memory
2. Load original images (left and right) to GPU global memory
3. (If real-world image) Pre-process images with Sobel / Residual

4. Calculate data cost
5. Calculate the data (Gaussian) pyramid
6. Message passing using created pyramid
7. Compute disparity map from messages and data-cost
8. Retrieve disparity map to local (host) memory

For the case of image sequences (as for driver assistance applications), Step 1 only needs to be done once. After this, all images can be processed using this memory allocation, thus start at Step 2 (saving computational time). These details are elaborated below. Example output of the algorithm described above can be seen in Figure 3.

The important limitations on the GPU alter the BP GPU kernels. These are the maximum thread limit $T_{\max}$ ($= 512$ on the GTX 280), and the shared memory S ($= 16$ KB of memory, which is 4096 single precision 32-bit float data). Keeping functions within the shared memory limit is what makes CUDA programming fast, so we try to keep to these limitations.

The final point to note is that with CUDA programming, you need to divide the threads into j (rows) by k (columns) blocks (omitting the 3rd dimension of 1). You also need to define the q (rows) by r (columns) grid. This defines how the images (M rows by N columns), within each kernel, are processed.

Allocate Global Memory. Memory is allocated and set to zero as follows (all data is single precision, i.e. 32-bit float):

– Left and right image: $2 \times MN$
– Preprocessed left and right images: $2 \times MN$
– Data Pyramid for message passing (central pixel plus: left, right, up and down): $2 \times 5 \times nMN$, where the "$2\times$" part is the upper bound (sum to infinity)
– Disparity image: $1 \times MN$

From this it can be seen that the memory allocation used is $(5 + 10n)MN$ float values. Note: This provides an upper bound on memory allocation dependent on M, N, and n. The limitations of the GTX 280 is 1 GB memory, so we can handle up to around $M = 600$, $N = 800$, and $n = 50$, which is larger than the general resolution in DAS (usually VGA $M = 480$, $N = 640$, and $n = 70$). Since all data is floating point, $S = 4096$ for the remainder of this section (based on the GTX 280).

Load Images to Global Memory. Load the left and right images to allocated memory and smooth with a Gaussian filter of $\sigma = 0{\cdot}7$ before computing the data costs.

Pre-Processing of Images. The images are pre-processed using either Sobel (as done in [10]) or residual images [16]. This is simply done by splitting the image into $j \times k$ overlapping windows. j and k are equal to the height and width (respectively) of the window used in each process. This allows massive parallel windows to be used. The typical window size is $j = 3$ and $k = 3$ (for both Sobel and residual) and only the input, output and two sets of working data are required for both Sobel (gradient in both directions) and one set for residual (smoothed image). This means that we are well within the limits of threads and shared memory, $4jk \leq T_{\max} \leq S$.

Calculate Data Cost. This part calculates the data cost for the image; this is a function to calculate the data cost at level $L = 0$ (i.e., D^0). Since we are dealing with only scanline specific information, the image is divided into a grid of $r = \left\lceil \frac{N}{T_{\max}+n} \right\rceil$ (need to store the maximum disparity and the scanline) times $q = M$. Each block will be $j = 1$ times $k = \left\lceil \frac{N}{r} \right\rceil + n$ large, which is under the $T_{\max}$ limit. This means that the total number of blocks is rM. Furthermore, only three data are used (left image, right image, and data-cost image), so $3k \leq 3T_{\max} \leq S$ (in the GTX 280). This means that the shared memory is exploited for fast computation. Comparing this to a conventional CPU, the number of cycles drops from nMN to effectively nrM.

Calculate Gaussian Pyramid. Here we calculate the data pyramid (using the data cost calculated above) in parallel. We employ L levels in the data pyramid to compute the messages and obtain D^1 to D^L. This is done by splitting the image at level $l \in L$ such that the $jk \leq T_{\max}$ limit is satisfied. Only the data from level l and $l + 1$ is needed, so the memory requirement is $2jk$, which is always in the shared memory limit $2jk \leq 2T_{\max} \leq S$. As long as the above limit is kept, any jk can be chosen. A number is ideal that exploits the size of the image (splitting it into even sized blocks) to not waste any threads (i.e., q, r, j, and k are all integers without overlap).

Message Passing. The implementation of the BP algorithm from coarse to fine in L levels is processed here. Here we divide the data cost D^l to a series of tiles whose size is $j \times k$; this is shown in Figure 4. These tiles are overlapping, and the image *apron* (image plus padding from this overlap) is formed. Since both j and k need to have to be odd, we define $j = (2\hat{j} + 1)$ and $k = (2\hat{k} + 1)$, respectively, to represent this. Obviously, one requirement is $jk \leq T_{\max} \Rightarrow (2\hat{j} + 1)(2\hat{k} + 1) \leq T_{\max}$. Since the message passing requires five data pyramids (centre, left, right, up and down pixel) at each disparity, the amount of memory needed is $5jkn$. If we want to exploit the shared

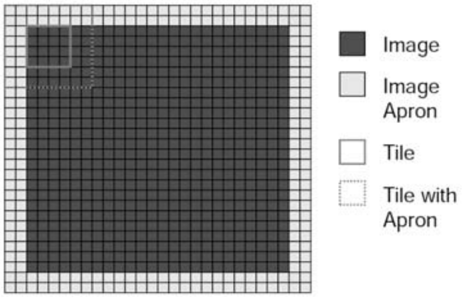

Fig. 4. Image *apron*. This shows how the image is segmented for message passing. Overlapping windows are required, and thus a padding is required to form the *apron*.

memory, the second requirement is: $5jkn \leq S \Rightarrow 5n(2\hat{j}+1)(2\hat{k}+1) \leq S$. One last requirement is that $j, k \geq 3 \Rightarrow \hat{j}, \hat{k} \geq 1$. Summarising these requirements:

$$(2\hat{j}+1)(2\hat{k}+1) \leq T_{\max} \tag{2}$$

$$5n(2\hat{j}+1)(2\hat{k}+1) \leq S \tag{3}$$

$$1 \leq \hat{j}, \hat{k} \in \mathbb{Z} \tag{4}$$

Limit (3) suggests that the maximum disparity calculation is in fact $n \leq \frac{S}{45}$, thus the maximum disparity on the GTX 280 is $n = 91$. If the disparity is above this, then shared memory is not used, thus Limit (3) is no longer enforced.

The grid requires overlap, thus $q = \left\lceil \frac{M^l}{j-2} \right\rceil$ and $r = \left\lceil \frac{N^l}{k-2} \right\rceil$, where M^l and N^l are the rows and columns, respectively, of the data at level $l \in L$.

Compute Disparity Map. Retrieve the estimated disparity map by finding, for each pixel in a block, the disparity that minimizes the sum of the data costs and message values. This process has the same limitations as the message passing for defining the block and grid sizes. However, there is one more piece of memory required (the disparity map), thus Limit (3) becomes $(5n+1)(2\hat{j}+1)(2\hat{k}+1) \leq S$. This reduces the maximum disparity to $n = 90$.

Retrieve Disparity Map. Transport the disparities from the GPU back to the host.

4 Experimental Results

In our experiments we compare high-contrast indoor images (Tsukuba), to real-world images (from HAKA1) which contain various types of noise, such as changes in lighting, out-of-focus lenses, differing exposures, and so forth. So we take two alternative measures to remove this low frequency noise, the first one is using the Sobel edge operator before BP [10], and another method is using the concept of residuals, which is the difference between an image and a smoothed version of itself [16]. Examples of both the Sobel and residual images for real-world images are shown in Figure 5. We use the BP algorithm outlined above with five levels (i.e., $L = 4$) in the Gaussian pyramid, and seven iterations per level.

For the Tsukuba stereo image pair, we have $M = 288$ and $N = 384$; the BP algorithm was run using a data cost of T_{data} = 15·0, discontinuity cost of T_{disc} = 1·7, and smoothness parameter of λ = 0·07, and the disparity space runs from 0 to 15 (i.e., $n = 16$). The only other parameters that need to be set explicitly are $\hat{j}, \hat{k} = 3$ (to fit into the shared memory). Figure 3 shows the results of running the implementation on two stereo pairs of images.

The real-world images recorded with HAKA1 are 640×480 ($N \times M$) with an assumed maximum disparity of 32 pixels (i.e., $n = 33$). We use $\hat{j}, \hat{k} = 2$ to fit into shared memory. The BP parameters are T_{data} = 30·0, T_{disc} = 11·0, and λ = 0·033. Example results for the HAKA1 sequences, with and without pre-processing, are shown in Figures 3, 5, and 6.

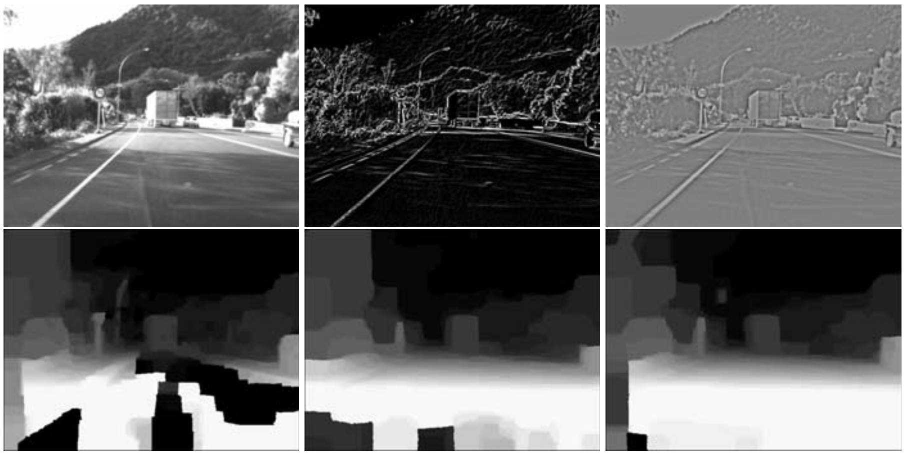
Fig. 5. Top row shows the images (left to right): original, after Sobel processing, mean-residual image. Bottom row shows the disparity results on the images in the top row.

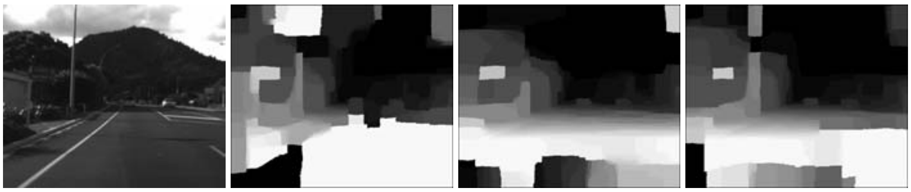
Fig. 6. Left to right: original image and disparity results, first on original images, then Sobel, and finally on residual images

We tested the speed of BP on the NVIDIA Geforce GTX 280 with CUDA compared to the normal PC (outlined in Section 2). The normal CPU implementation runs at 32·42 seconds for the Tsukuba stereo pair, while the running time using CUDA is 0·127 seconds, a speed up by a factor of over 250! The time of BP for real-world images, when running on the normal PC, is 93·98 seconds, compared to 2·75 seconds using the CUDA implementation, and this is (only) a speed improvement by factor 34.

5 Conclusions

We have implemented the belief propagation algorithm (applied to stereo vision) on programmable graphics hardware, which produces fast and accurate results. We have included full details on how to run and implement belief propagation on CUDA. We divided the stereo pairs to a lattice in order to suit the architecture of a GPU. Results for real images are not satisfying as on high-contrast indoor images without pre-processing. Resulting disparity images improve by using either the Sobel edge operator or residual images as input, as suggested in prior work. We have also defined the limitations of this type of BP implementation using limiting inequalities and suggestions for appropriate numbers.

Future work will aim at better speed improvement by exploiting the texture memory. Other ideas for speed improvement are to initialise the BP algorithm with Dynamic Programming Stereo on CUDA, thus speeding up convergence. Finally, the inequalities specified in this paper are ideal for implementation into a linear (quadratic integer) programming optimisation scheme to choose the optimal parameters according to image size.

Acknowledgement

This work is supported by the Natural Science Found of China under No. 60872119.

References

1. CUDA Zone, `http://www.nvidia.com/cuda`
2. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut / max-flow algorithms for energy minimization in vision. IEEE Trans. Pattern Analysis Machine Intelligence 26, 1124–1137 (2004)
3. Brunton, A., Chang, S., Gerhard, R.: Belief Propagation on the GPU for Stereo Vision. In: Proc. Canadian Conf. Computer Robot Vision, p. 76 (2006)
4. *.enpeda..* image sequence analysis test site (EISATS), `http://www.mi.auckland.ac.nz/EISATS/`
5. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. Int. J. Computer Vision 70, 41–54 (2006)
6. Fung, J., Mann, S., Aimone, C.: OpenVIDIA: Parallel GPU computer vision. In: Proc. of ACM Multimedia, pp. 849–852 (2005)
7. Fung, J., Mann, S.: Using graphics devices in reverse: GPU-based image processing and computer vision. In: Proc. IEEE Int. Conf. Multimedia Expo., pp. 9–12 (2008)
8. Govindaraju, N.K.: GPUFFTW: High performance GPU-based FFT library. In: Supercomputing (2006)
9. Grauer-Gray, S., Kambhamettu, C., Palaniappan, K.: GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In: Proc. PRRS, pp. 1–4 (2008)
10. Guan, S., Klette, R., Woo, Y.W.: Belief propagation for stereo analysis of night-vision sequences. In: Wada, T., Huang, F., Lin, S. (eds.) PSIVT 2009. LNCS, vol. 5414, pp. 932–943. Springer, Heidelberg (2009)
11. Klette, R.: Analysis of data flow for SIMD systems. Acta Cybernetica 6, 389–423 (1984)
12. NVIDIA. NVIDIA CUDA Programming Guide Version 2.1 (2008), `http://www.nvidia.com/object/cuda_develop.html`
13. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. Computer Vision 47, 7–42 (2002)
14. Sinha, S.N., Frahm, J.M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using graphics hardware. In: Proc. Machine Vision and Applications (2006)
15. Vineet, V., Narayanan, P.J.: CUDA cuts: Fast graph cuts on the GPU. In: CVPR Workshop on Visual Computer Vision on GPUs (2008)
16. Vaudrey, T., Klette, R.: Residual Images Remove Illumination Artifacts for Correspondence Algorithms! In: Proc. DAGM (to appear, 2009)
17. Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., Nistér, D.: Real-time global stereo matching using hierarchical belief propagation. In: Proc. British Machine Vision Conf., pp. 989–998 (2006)