Belief Propagation Implementation using CUDA on XPS

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 report describes an efficient implementation of a belief propagation algorithm on GPUs that can be used to speed up stereo image processing. 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.

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 [4].

Implementations of 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 [14]. Ideally, one wants to combine the accuracy achieved via global methods with the running time of local methods. One option towards 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 the BP algorithm using the NVIDIA Compute Uniform Device Architecture (CUDA) language environment. 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* [15] implemented a feature based tracker on the GPU. SiftGPU [18] implements the SIFT descriptor on the GPU. Recently, Vineet [16] 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 report explains the implementation details clearly. We then go on to detail important pre-processing steps that can be done to improve results on real-world data (with real-world noise).

The report 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 Used Processor and Test Data

We use either a normal PC (Intel Core 2 Duo CPU running at 2.13 GHz and 3 GB memory) or a GPU on an XPS machine. 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 $1980s$; 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

General purpose programming on graphics processing units (GPGPU) tries to solve a problem by posing it as a graphics rendering problem, restricting the range of solutions that can be ported to the GPU [16]. The GPU memory layout is optimized for graphics rendering. This restricts GPGPU solutions to be possibly not available for optimal data structures. The GPGPU model provides limited autonomy to individual processors. Creating efficient data structures using the GPU memory model is a challenging problem in itself [12]. The available memory on a GPU is another restricting factor. A single data structure on the GPU cannot be larger than the maximum texture size supported by it.

Compute Unified Device Architecture (CUDA) is a programming interface that uses the parallel architecture of NVIDIA GPUs for general purpose

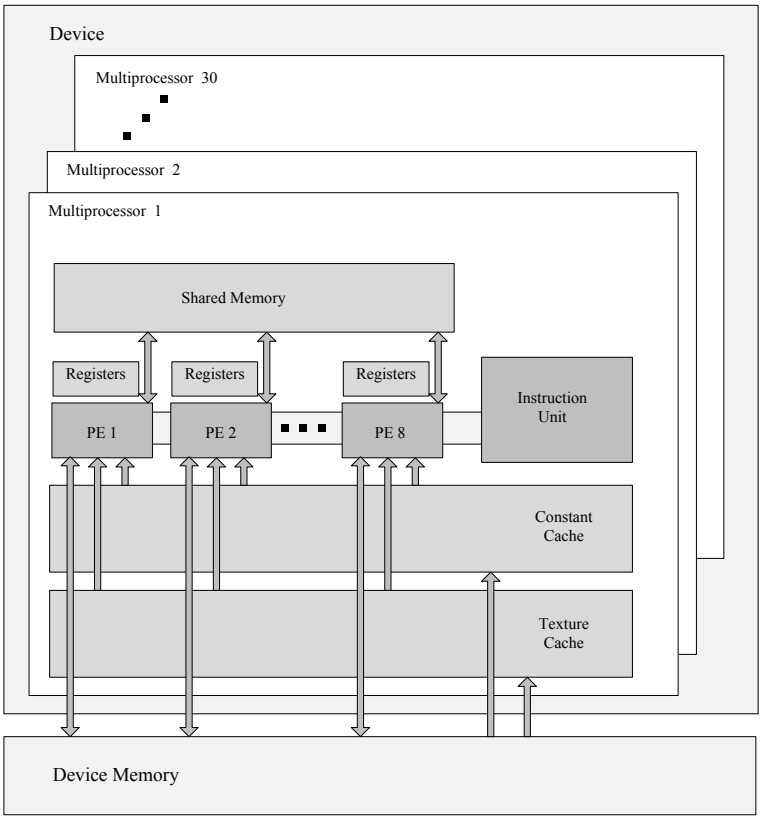

Fig. 1. CUDA Hardware Model for NVIDIA GeForce GTX 280.

computing. CUDA produces a set of library functions as extensions of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. All memory available on the device can be accessed using CUDA with no restrictions on its representation though access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the GPU for general purpose computing.

CUDA Hardware Model. At the hardware level, the GPU is a collection of multiprocessors, with several processing elements (PEs) in each of them (Figure 1) [13]. For example, the NVIDIA GeForce GTX 280 has 30 multiprocessors, each with 8 processing elements. Each multiprocessor has 16 KB of common shared memory accessible to all PEs inside of it. It also has a set of 32-bit registers, texture, and constant memory caches. All PEs in all multiprocessors execute the same instruction at a given cycle. Each PE can

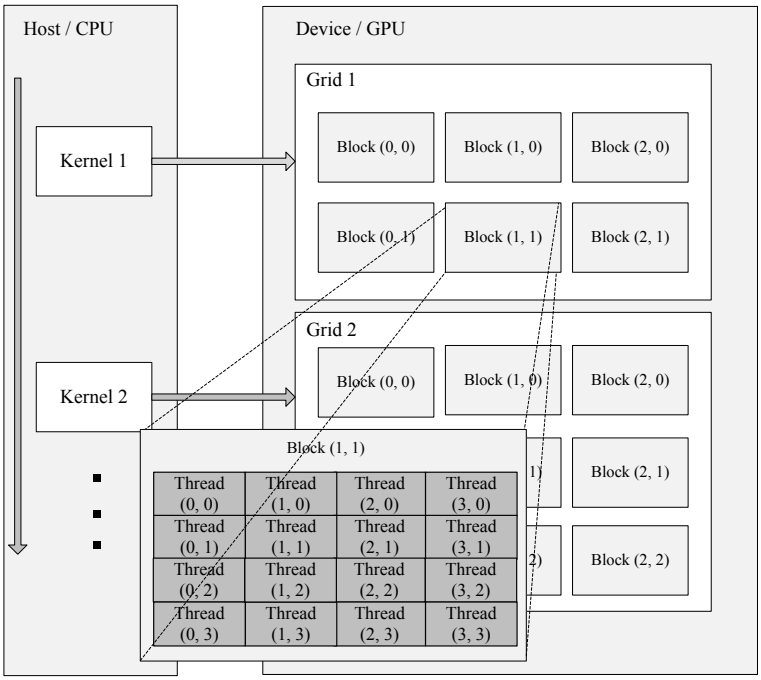

Fig. 2. CUDA Programming Model for NVIDIA GeForce GTX 280.

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 PEs of all the multiprocessors. The PEs of a multiprocessor can synchronize with one another, but there is no direct synchronization mechanism between the multiprocessors.

CUDA Programming Model. For the programmer, the CUDA consists of a collection of threads running in parallel. 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. The programmer can select the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads, called a *block*, runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor for time-shared execution. They also divide the common resources like registers and shared memory equally among them. A single execution on a device generates a number of blocks. The collection of all blocks in a single execution is called a *grid* (Figure 2).

Each thread and block is given a unique ID that can be accessed within the thread during its execution. All threads of the grid execute a single program called the *kernel*. The kernel is the core code to be executed on each thread.

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, which can access any memory location.

The CUDA programming interface represents a Parallel Random Access Machine (PRAM) architecture if one uses the device memory alone. The performance improves with the use of shared memory which can be accessed in a single clock cycle. In contrast, the global or device memory access takes 200–400 cycles [13].

Read-only texture memory, optimized for 2D texture, fetch and constant memory assigned by the CPU are also available. Their access is slow but the internal caching mechanism reduces the effective access times for coherent access. The hardware architecture allows multiple instruction sets to be executed on different multiprocessors. The current CUDA programming model, however, cannot assign different kernels to different multiprocessors, though this may be simulated using conditionals.

The NVIDIA GeForce GTX 280 graphics card has 1 GB memory. Large images can reside in this memory, given a suitable representation. The problem needs to be partitioned appropriately into multiple grids for handling even larger images and graphs.

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 (Figure 3), and real-world stereo sequences, which are captured with HAKA1 (Figure 4), a research vehicle of the *.enpeda..* project at The University of Auckland [3].

3 Implementation of Belief Propagation

Our GPU implementation is based on the "multiscale" BP algorithm presented by Felzenszwalb and Huttenlocher [4]. If run on the original stereo images, it

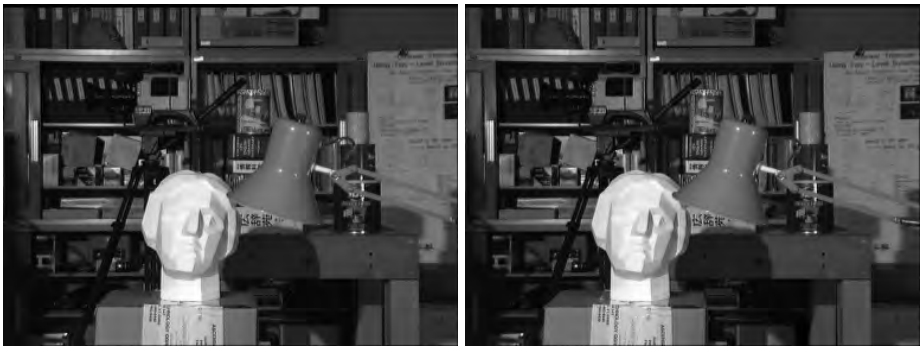

Fig. 3. Tsukuba stereo pair [14].

Fig. 4. Real image pair captured by HAKA1.

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 [17] provides a general study (i.e., for the use of residual input images).

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

$$f\colon P \to L \tag{1}$$

(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 [1]) as follows:

$$E(f) = \sum_{p \in P} D_p(f_p) + \sum_{(p,q) \in A} V(f_p - f_q), \tag{2}$$

$D_p(f_p)$ is the cost of assigning label f_p to pixel p, and 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.

Since we aim at minimizing the energy, this approach corresponds to the Maximum A-Posteriori (MAP) estimation problem. In the stereo vision case, the data cost $D_p(f_p)$ may be defined by the difference in intensities between pixels in left and right image defined to be 'corresponding' when applying disparity f_p. The discontinuity cost $V(f_p - f_q)$ increases as the difference between f_p and f_q increases. The goal is to find the disparity image that minimizes the total energy $E(f)$. This problem is equivalent to finding the maximum a posteriori (MAP)

estimator of a MRF [4], so an approximate solution can be found using the loopy BP algorithm that is used for inference on MRFs.

The BP algorithm is defined by iterations of passing a message around in the image grid. Typically, 4-adjacency is assumed. The message update schedule is in each iteration, and messages pass from node to node in parallel. Then the values in the received messages and the data costs are used to compute the messages to send in the next iteration. Each message is represented as an array, which size is determined by the maximum disparity. After all the iterations are complete, the data costs and the values for each label in the messages are used to retrieve the estimated disparity at each pixel.

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.

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. (Such a red-black technique is also used for Gauss-Seidel relaxations. A Gauss-Seidel relaxation attempts to increase the convergence rate by using values computed for the kth iteration in subsequent computations of the kth iteration.)

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 common data pyramid is used: Nodes in one 2×2 array in one layer are also adjacent to one node in the next layer. All $2^m \times 2^m$ pixels at the bottom layer are connected this way with a single node at top layer m. 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: [2] and [19] describe GPU implementations of BP on a set of stereo images. However, each of these implementations use 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 GPUs to implement the generation of disparity images with the BP algorithm. The first kernel is calculating the data cost in parallel, then another function is used to

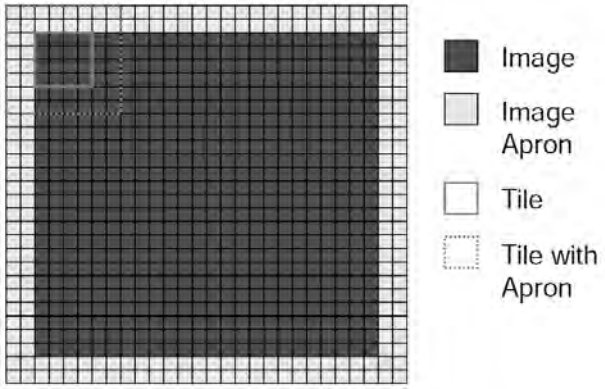

Fig. 5. Tile apron.

implement the Gaussian pyramid for five levels. The third one is for computing messages with 7 iterations, and finally we define the last kernel function for the output of the disparity image. In the implementation of computing messages, which takes most of the time, a pixel is processed by one thread in the block. For the Tsukuba images, where the size equals 384×288, 5×5 pixels are computed at the same time in one block with 7×7 threads in each. The blocks in the kernel function (a grid) are dependent on the level of computing. For the first level, 77×58 blocks are used, and 39×29 in the second level.

In detail, the implementation contains the following steps:

1. Allocate memory on the global memory of GPUs for stereo pairs, data cost and messages with the messages initialized by zero.
2. Load left and right images to the host, and smooth with a Gaussian filter of $\sigma = 0{\cdot}7$ before computing the data costs, then transport data from host to GPUs.
3. Compute data cost. After that we call the first kernel function to calculate the data cost D_0, with r blocks (number of rows which is 288 for the Tsukuba image) and s threads (which is the number of columns plus maximum disparity, i.e., 400 for Tsukuba) in each block. Each block in the GPU contains automatically synchronized threads. Maximum disparity is 16 for the Tsukuba image. For improving the speed, all pixels in each row are processed in parallel. When calculating cost of each pixel, 16 more threads are needed for each row. In this kernel, we just use 288 cycles instead of 384×288 cycles on a CPU.
4. Calculate the data pyramid in parallel with the second kernel function. We employ five levels in the data pyramid to compute the messages and obtain D_1 to D_{L-1} here.
5. Implement BP algorithm from coarse to fine in five levels with the third kernel function. Here we divide the data cost D_p to a series of tiles which

Fig. 6. Residual images of a scene recorded with HAKA1.

size is $k \times j$ (which are the pixels we calculate in one block, i.e., 5×5 for Tsukuba), so that the smaller tile can be cached to the shared memory of the block (shown in Figure 5). As this can not be synchronized for different blocks, we must transport more pixels (which we name *apron* here, and those are 7×7 for Tsukuba) for calculation messages with BP. After receiving message for every pixel in parallel, we send them to adjacent pixels using the current message values and data costs. Finally, we obtain messages at the highest level.

6. Get the disparity image using the last kernel function. 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.
7. Last step: transport the disparities from the GPU back to the host.

For a stereo pair sequence, we just need to allocate memory on the GPU (Step 1) once for the first pair, and set all that memory to zero after transporting the result back to the host. Then we can process the next stereo pair beginning at Step 2, which will save more time. All of these steps have also been implemented in [9], but the details of computing data costs or updating messages in parallel are not made clear in [9]. In comparison, we also achieved a faster implementation.

Compared to high-contrast indoor images such as the Tsukuba stereo pair, real-world images 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 [17]. The residuals of real-world images are shown in Figure 6.

4 Experimental Results

The BP algorithm that was implemented in CUDA is based on the algorithm introduced in [5] and the resulting GPU-based disparity images were validated

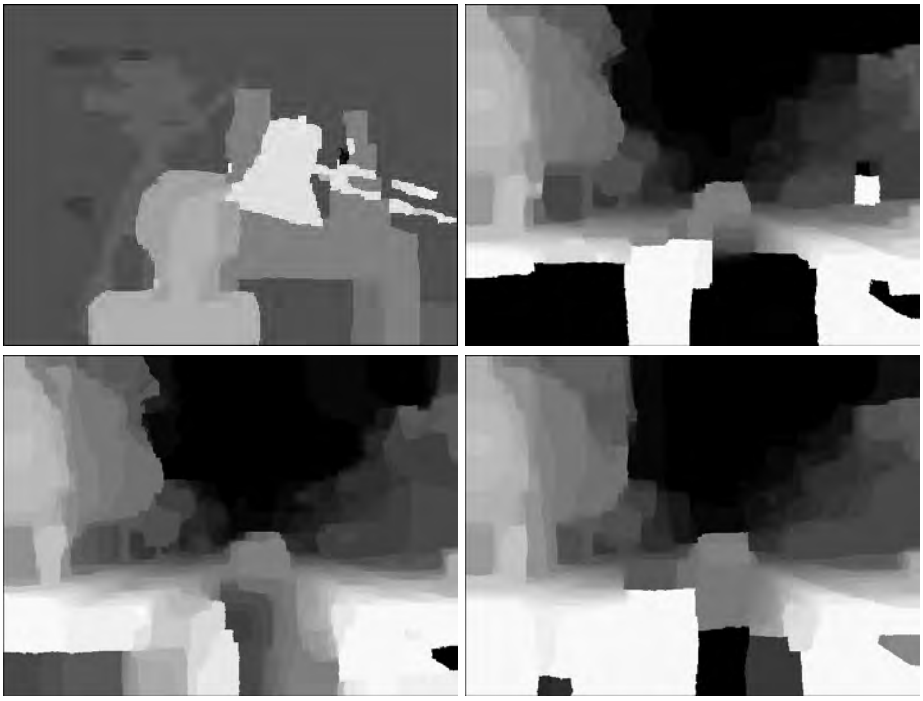

Fig. 7. Upper left: disparity image for the Tsukuba stereo pair. Results for a HAKA1 sequence: disparity image for the original stereo pair (upper right), disparity image after applying a Sobel operator (lower left), disparity image calculated for the mean residual stereo pair (lower right).

to be nearly identical with the sequential version. Figure 7 shows the results of running the implementation on two stereo pairs of images. The Tsukuba stereo images are 384×288 with $T_{data} = 15{\cdot}0$, $T_{disc} = 1{\cdot}7$, $\lambda = 0{\cdot}07$, and the disparity space runs from 0 to 15. We use $r = 288$ blocks and $s = 400$ threads per block for computing data cost, and calculate 5×5 pixels in BP (Step 5) using 7×7 pixels. The real-world images recorded with HAKA1 are 640×480 with an assumed maximum disparity of 32 pixels. We use $r = 480 \times 2$ blocks with $s = 320 + 32$ threads per block, and calculate 3×3 pixels in BP (Step 5) using 5×5 pixels. (to keep within physical limits of the 280 GTX). The parameters are $T_{data} = 30{\cdot}0$, $T_{disc} = 11{\cdot}0$, and $\lambda = 0{\cdot}033$. The BP implementations of both stereo pairs uses 5 levels, and 7 iterations per level.

We test the speed of belief propagation on the NVIDIA Geforce GTX 280 with CUDA and a normal PC. The normal PC is an Intel Core 2 Duo CPU running at 2·13 GHz and 3 GB memory. The CPU implementation runs 32·42 seconds on the normal PC for the Tsukuba stereo pair, while the running time using CUDA is 0·127 seconds when the time to transfer image data to the GPU

and the disparity values back to the host is included, and 0·081 milliseconds when the transfer time is not included. The time of real-world images (Figure 4)

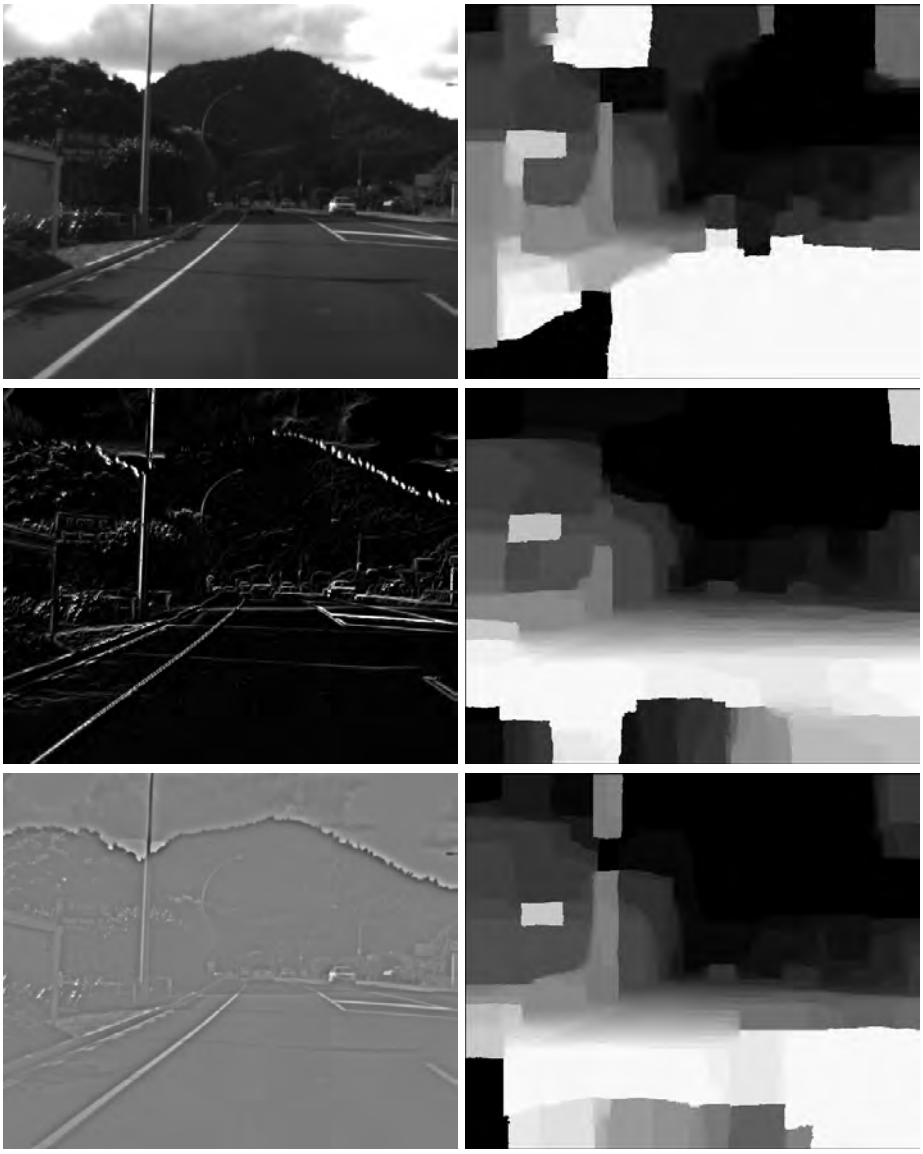

Fig. 8. Upper left: original image captured by HAKA1. Upper right: disparity image for the original stereo pair. Middle left: stereo pair after Sobel operator. Middle right: disparity image after applying a Sobel operator. Lower left: stereo pair after residual. Lower right: disparity image calculated for the mean residual stereo pair.

running on normal PC is 93·98 seconds, compared with 2·75 seconds using CUDA implementation, and its running time without transfer time is 0·115 seconds.

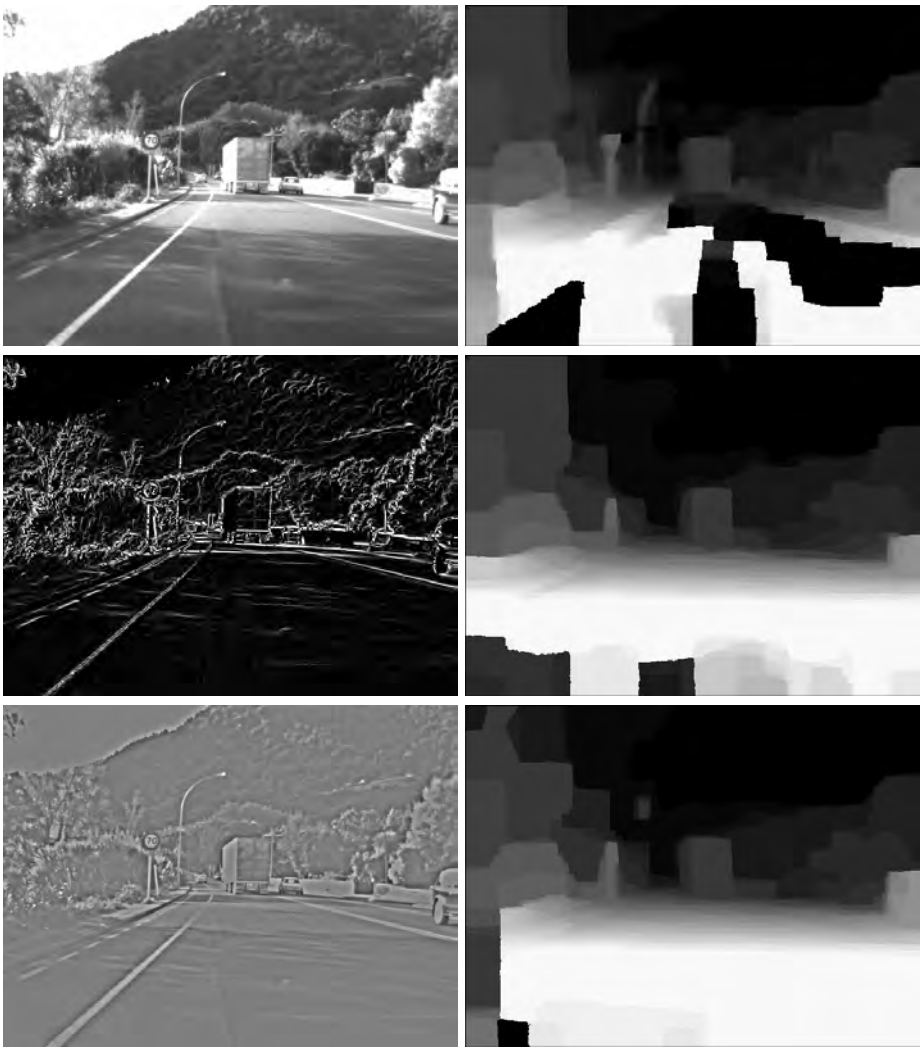

Fig. 9. Upper left: original image captured by HAKA1. Upper right: disparity image for the original stereo pair. Middle left: stereo pair after Sobel operator. Middle right: disparity image after applying a Sobel operator. Lower left: stereo pair after residual. Lower right: disparity image calculated for the mean residual stereo pair.

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 (shown in the upper right image of Figure 7) are not satisfying as on high-contrast indoor images (shown in the upper left image of Figure 7). Resulting disparity images improve by using either the Sobel edge operator or residual images as input, as suggested elsewhere. We still aim at a better speed improvement to achieve real-time processing in driver-assistance applications; we will continue optimizing the implementation of the BP algorithm, possibly by utilizing an initialization using Dynamic Programming Stereo on CUDA. Also, exploiting the texture memory has not yet been discussed, and this is an idea for future improvements.

References

1. Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut / max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis Machine Intelligence*, 26:1124–1137, 2004.
2. A. Brunton, S. Chang and R. Gerhard. Belief Propagation on the GPU for Stereo Vision. In Proc. *3rd Canadian Conf. Computer and Robot Vision*, page 76, 2006.
3. *.enpeda..* image sequence analysis test site (EISATS). http://www.mi.auckland.ac.nz/EISATS/
4. P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. In Proc. *CVPR*, volume 1, pages 261–268, 2004.
5. P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. *Int. J. Computer Vision*, **70**:41–54, 2006.
6. J. Fung, S. Mann and C. Aimone. OpenVIDIA: Parallel GPU computer vision. In *Proc of ACM Multimedia 2005*, pages 849–852, 2005
7. J. Fung and S. Mann. Using graphics devices in reverse: GPU-based image processing and computer vision. In Proc. *IEEE Int. Conf. Multimedia Expo*, pages 9–12, 2008.
8. N. K. Govindaraju. GPUFFTW: High performance GPU-based FFT library. In *Supercomputing*, 2006
9. S. Grauer-Gray, C. Kambhamettu and K. Palaniappan. GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In Proc. *PRRS2008*, pages 1–4, 2008
10. S. Guan, R. Klette, and Y.W. Woo. Belief propagation for stereo analysis of night-vision sequences. In Proc. *PSIVT*, LNCS 5414, pages 932–943, 2009.
11. R. Klette. Analysis of data flow for SIMD systems. *Acta Cybernetica*, **6**:389–423, 1984.
12. A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. In Proc. *ACM Trans. Graph.*, 25(1):60–99, 2006
13. *NVIDIA.* NVIDIA CUDA Programming Guide Version 2.1, 2008 http://www.nvidia.com/object/cuda_develop.html

14. D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Computer Vision*, **47**:7–42, 2002.
15. S. N. Sinha, J. M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using graphics hardware. In *Proc of Machine Vision and Applications*, 2006.
16. V. Vineet and P. J. Narayanan. CUDA Cuts: Fast Graph Cuts on the GPU. In: *CVPR Workshop on Visual Computer Vision on GPUs*, 2008
17. T. Vaudrey and R. Klette. Residual Images Remove Illumination Artifacts for Correspondence Algorithms! Technical report, MItech-TR-35, http://www.mi.auckland.ac.nz/, University of Auckland (2009)
18. C. Wu and M. Pollefeys. Siftgpu library. Technical Report http://cs.unc.edu/~ccwu/siftgpu/, UNC, Chapel Hill, 2005.
19. Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nistér. Real-time global stereo matching using hierarchical belief propagation. In *British Machine Vision Conf.*, pages 989–998, 2006.

Some Useful Links and Notes

Links

1. CUDA, Supercomputing for the Masses. http://www.ddj.com/cpp/207200659
2. CUDA Zone. http://www.nvidia.com/object/cuda_home.html
3. CUDA Stereo. http://www.cs.unc.edu/~gallup/stereo-demo/

Useful Notes

1. The maximum size of shared memory we can use in a block is only 64 KB (64×64 float data).
2. When we compute messages during BP on the GPU, the maximum number of pixels which can be processed in a block depends on the maximum disparity of the stereo pairs.