Mutual Information based Semi-Global Stereo Matching on the GPU

Ines Ernst[1] and Heiko Hirschmüller[2]

[1] Department of Optical Information Systems, Institute of Robotics and Mechatronics, German Aerospace Center (DLR), Berlin (Germany), ines.ernst@dlr.de
[2] Department of Robotic Systems, Institute of Robotics and Mechatronics, German Aerospace Center (DLR), Oberpfaffenhofen (Germany), heiko.hirschmueller@dlr.de

Abstract. Real-time stereo matching is necessary for many practical applications, including robotics. There are already many real-time stereo systems, but they typically use local approaches that cause object boundaries to be blurred and small objects to be removed. We have selected the Semi-Global Matching (SGM) method for implementation on graphics hardware, because it can compete with the currently best global stereo methods. At the same time, it is much more efficient than most other methods that produce a similar quality. In contrast to previous work, we have fully implemented SGM including matching with mutual information, which is partly responsible for the high quality of disparity images. Our implementation reaches 4.2 fps on a GeForce 8800 ULTRA with images of 640×480 pixel size and 128 pixel disparity range and 13 fps on images of 320×240 pixel size and 64 pixel disparity range.

1 Introduction

Fast stereo matching is necessary for many practical real-time applications, including robotics. Often, it is not necessary to perform stereo matching at the video frame-rate. Instead, processing several frames a second in a VGA like resolution can be sufficient.

Commercial real-time stereo systems are either based on special hardware that delivers disparity images at frame-rate [1, 2] or are available as pure software development kit [3]. These solutions are all based on local approaches that perform correlation of rectangular windows, followed by a winner-takes-all disparity selection. Correlation methods are fast, but they are known to blur object boundaries and remove small structures [4]. More elaborate methods require higher computational resources, which are available on graphics cards.

Modern graphics processing units (GPU) are usable as high-speed coprocessors for general purpose computational tasks. For several years already, high-end graphics processors have been supporting high performance applications through dedicated programmable vertex and fragment processors [5]. However, programs on these GPUs were limited to the capabilities of the specialized hardware. With only a few exceptions [6] the existing graphics APIs required the transformation of computationally intensive core algorithms into rendering tasks. In 2007, NVIDIA introduced with the G8 series a new generation of GPUs [5]. The Compute Unified Device Architecture (CUDA) [7] combines a new hardware concept (built around just one type of programmable

4th International Symposium on Visual Computing (ISVC08), Las Vegas, NV, USA, 1-3 December 2008

processor) with a new and more flexible programming model. CUDA provides a C-like abstraction layer for implementing general purpose applications on GPUs without substantial knowledge of underlying hardware or graphics concepts. CUDA cards overcome limitations of earlier hardware such as unsupported data scattering on fragment shader level, memory read- or write-only properties or missing integer arithmetic. The CUDA concept is continued with the introduction of the G9, G200 and TESLA series with ever growing numbers of unified processors and amounts of on-board memory[3]. Nevertheless, all new GPUs continue to support the widely used graphics APIs OpenGL and DirectX for their original task, i.e. very fast graphics rendering.

2 Previous Work

There are real-time GPU implementations of local stereo methods that try to increase accuracy by different aggregation methods [8]. Others join multiple resolutions [9] or windows [10]. Plane sweep stereo methods appear particularly well suited for GPU implementations and reach high frame rates [11–13].

None of the stereo methods above are included in the Middlebury online evaluation [14]. In fact, almost all methods of this comparison are global stereo approaches that perform pixelwise matching, controlled by an energy function that connects all pixels of the image with each other. Global methods are more accurate than local methods [15], but their internal complexity is typically much higher than the complexity of local methods. Therefore, their run-time is often several orders of magnitudes higher than that of local methods, which makes them unsuitable for real-time applications.

An exception are dynamic programming solutions, which are global in one dimension. Gong and Yang [16] reached real-time performance with a two pass dynamic programming method, implemented on the GPU. The average error (i.e. average of errors at non-occluded, all and discontinues pixel areas over four datasets) is 10.7% according to the Middlebury evaluation.

Another exception is the Semi-Global Matching (SGM) method [17], which combines several one-dimensional optimizations from all directions. Its complexity is $O(width \times height \times disparityrange)$, like local methods, which results in efficient computations. On the other hand, its accuracy is comparable to that of global methods. The Middlebury evaluation rates the original method with 7.5%, while the consistent variant for structured environments has an average error of just 5.8%. This is not far away from the average error of 4.2% of the currently very best method in this evaluation. When looking at the run-times, even the CPU implementation of SGM is several orders of magnitudes faster than most other methods of the evaluation. Furthermore, the algorithm has a regular structure and the basic operations are very simple, which allows an efficient GPU implementation.

Rosenberg et al. [18] implemented the core part of SGM on a NVIDIA 7900 GTX using Cg. Their implementation includes matching with absolute differences, left/right consistency checking and hole filling. The implementation reached 8 fps on images of 320×240 pixel size with 64 pixel disparity range. Gibson and Marques [19] used

[3] Developments of other GPU vendors are not reviewed within this article.

CUDA on a NVIDIA Quadro FX5600[4]. In their approach, matching is implemented with the sampling-insensitive absolute difference [20] and the smoothness penalty is adapted to the intensity gradient, but no left/right consistency checking is done. The implementation reaches 5.9 fps on images of size 450×375 pixels with 64 pixel disparity range.

It has been found that matching with mutual information performs better than absolute differences, even on images with no apparent radiometric differences [21]. Furthermore, the hierarchical computation of mutual information (HMI) gives the same result as the iterative computation [17]. Therefore, in contrast to previous work, we implemented the full SGM method on the GPU, including hierarchically calculated mutual information, intensity gradient sensitive smoothness penalty, left-right consistency checking and sub-pixel interpolation.

The SGM algorithm is reviewed in Section 3. Its implementation on the GPU is explained in Section 4. The quality and speed of the implementation are evaluated and compared to previous implementations in Section 5. Section 6 concludes the paper.

3 The Semi-Global Matching Algorithm

We assume a rectified, binocular stereo pair as input and refer to the 8 bit intensity values of the left and right image with I_L and I_R. The following sections describe the individual steps of the method from an algorithmic point of view, which is visualized in Figure 1. The interested reader is referred to the original publications [17] for the derivation and justification of these steps.

3.1 Pixelwise Matching Costs using Mutual Information

The cost for matching two pixels is derived from mutual information (MI). It is computed from the joint entropy of correspondences of both images (H_{I_L,I_R}) and the entropies of the left (H_{I_L}) and right image (H_{I_R}) as

$$MI_{I_L,I_R} = H_{I_L} + H_{I_R} - H_{I_L,I_R}. \tag{1}$$

The joint entropy, which is defined as $P\log(P)$, is computed by Taylor expansion, according to Kim et al. [22], as a sum of data terms h_{I_L,I_R} over all pixels $\mathbf{p}$ and their correspondences $\mathbf{q}$

$$H_{I_L,I_R} = \sum_{\mathbf{p}} h_{I_L,I_R}(I_{L\mathbf{p}}, I_{R\mathbf{q}}). \tag{2}$$

The data term is computed from the probability distribution (P) and convolution with a Gaussian kernel for Parzen estimation by

$$h_{I_L,I_R}(i,k) = -\frac{1}{n}\log(P_{I_L,I_R}(i,k) \otimes g(i,k)) \otimes g(i,k). \tag{3}$$

[4] According to personal communication with the author.

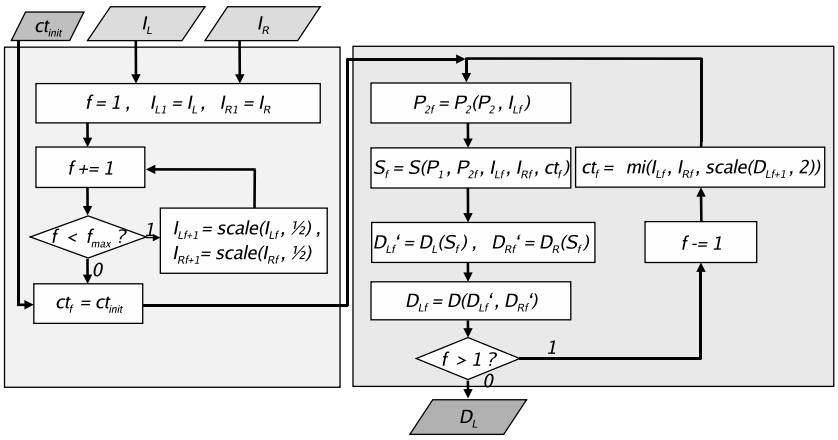


Fig. 1. Flowchart of the Semi-Global Matching method

The probability distribution is calculated from the histogram of corresponding intensities of both images. This requires an initial guess of correspondences, i.e. an initial disparity image D_{init}. Section 3.5 explains where this initial disparity image comes from. Thus, D_{init} is used for collecting corresponding intensities from I_L and I_R, by ignoring occlusions. Dividing all histogram entries by the number (n) of correspondences results in the joint probability distribution P_{I_L,I_R}, which is a table of 256×256 entries for 8 bit images. Thereafter, equation (3) is used for computing table h_{I_L,I_R} of data elements. The probability distributions P_{I_L} and P_{I_R} can be computed from P_{I_L,I_R} by simply summing over all lines or columns of P_{I_L,I_R}. The data terms h_{I_L} and h_{I_R} are computed similarly to (3), except that they are one dimensional arrays instead of a two dimensional table. Finally, all data terms are summed according to (1) for getting the table of matching costs

$$mi_{I_L,I_R}(i,k) = h_{I_L}(i) + h_{I_R}(k) - h_{I_L,I_R}(i,k). \tag{4}$$

It can be seen that summing with all pixel correspondences over this table results in the mutual information, which is to be minimized. For SGM, the table *mi* is used for computing the cost for matching pixel $\mathbf{p}$ with the corresponding pixels at disparity d

$$C(\mathbf{p},d) = mi_{I_L,I_R}(L(\mathbf{p}),R(\mathbf{p}-[d,0]^T)). \tag{5}$$

3.2 Aggregation of Pixelwise Matching Costs

The SGM method approximates the minimization of the energy,

$$E(D) = \sum_{\mathbf{p}}(C(\mathbf{p},D_\mathbf{p}) + \sum_{\mathbf{q}\in N_\mathbf{p}} P_1\,\mathrm{T}[|D_\mathbf{p} - D_\mathbf{q}| = 1] + \sum_{\mathbf{q}\in N_\mathbf{p}} P_2\,\mathrm{T}[|D_\mathbf{p} - D_\mathbf{q}| > 1]). \tag{6}$$

The first term sums the pixelwise matching costs for all pixels. The second term penalizes small discontinuities with a penalty P_1, while the third term penalizes all larger

discontinuities with a penalty P_2. The approximation is implemented by summing pathwise costs according to (6) into a cost volume. This process can be seen as aggregation. The pixelwise matching costs are aggregated into a cost volume ($S(\mathbf{p},d)$) by going in 8 directions (r) through all pixels of the image I_L. The directions are defined as $[1,0]^T$, $[1,1]^T$, $[0,1]^T$, $[-1,1]^T$ and so on. The first pixels of each path (i.e. the pixels at the image border) are defined by the pixelwise matching cost as $L_{\mathbf{r}}(\mathbf{p},d) = C(\mathbf{p},d)$. The costs at all further pixels in the path direction r are computed according to (6) by

$$\begin{aligned} L_{\mathbf{r}}(\mathbf{p},d) =& C(\mathbf{p},d) + \min(L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},d), L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},d-1)+P_1, \\ & L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},d+1)+P_1, \min_i L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},i)+P_2) - \min_i L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},i). \end{aligned} \tag{7}$$

The value of P_1 is a constant, while P_2 is adapted to the intensity gradient along the path by $P_2 = \frac{P_2'}{|I_{b\mathbf{p}} - I_{b\mathbf{p}-\mathbf{r}}|}$, with P_2' as constant. The values of $L_{\mathbf{r}}(\mathbf{p},d)$ are added to $S(\mathbf{p},d)$ for all disparities d at each pixel $\mathbf{p}$. Additionally, they are kept as previous values for the next step r along the path.

3.3 Disparity Selection

The disparity at each pixel is selected as the index of the minimum cost

$$D_L(\mathbf{p}) = argmin_d S(\mathbf{p},d). \tag{8}$$

Sub-pixel estimation is implemented by fitting a parabola through neighboring costs

$$D_{L\mathbf{p}}^{sub} = D_{L\mathbf{p}} + \frac{S(\mathbf{p},D_{L\mathbf{p}}-1) - S(\mathbf{p},D_{L\mathbf{p}}+1)}{2S(\mathbf{p},D_{L\mathbf{p}}-1) - 4S(\mathbf{p},D_{L\mathbf{p}}) - 2S(\mathbf{p},D_{L\mathbf{p}}+1)}. \tag{9}$$

This parabolic fitting is used as an approximation in the absence of a theoretically derived sub-pixel interpolation function for a complex matching cost like MI. However, it has been found that this choice delivers good results.

3.4 Post Filtering and Consistency Checking

The disparities that correspond to the right image are derived from the same cost volume S by a *diagonal* search for the minimum, i.e.

$$D_R(\mathbf{p}) = argmin_d S(\mathbf{p} + [d,0]^T, d). \tag{10}$$

Both disparity images are filtered with a $3x3$ median for removing outliers. The result is used for a left right consistency check. If $|D_{L\mathbf{p}} - D_{R\mathbf{q}}| > 1$ with $\mathbf{q} = \mathbf{p} - [d,0]^T$, then the value at $D_{L\mathbf{p}}$ is set to invalid.

3.5 Hierarchical Computation of MI

As discussed earlier, the computation of the matching cost table (4) requires an initial disparity image D_{init}. It has been found that MI can be computed hierarchically, by

starting with images that are downscaled by factor 2^f (e.g. 16), which results in I_L^f and I_R^f. The initial disparity is set to random values. The random sampling within the disparity range is sufficient for computing an initial matching table that is used for matching I_L^f and I_R^f, which results in the disparity image D_L^f. The disparity image is upscaled by simple interpolation to D_L^{f-1} and used as initial disparity image for matching I_L^{f-1} and I_R^{f-1}. The process is repeated until $f = 1$.

The hierarchical computation reduces the runtime of an otherwise iterative computation of MI that would require several runs at full resolution. It is important that the hierarchical computation is only used for refining the initial disparity for MI computation and not for reducing the disparity search range, as this could easily lead to loosing small objects. It has been found that the matching quality of the hierarchical computation of MI equals that of the iterative computation [17].

4 GPU Implementation

We started our work on the G7 GPU architecture for which the OpenGL/Cg [23, 24] programming technique is available. This decision was supported by the fact that important parts of the algorithm map very well to graphics concepts and the OpenGL drivers offer highly optimized parallelization and scheduling mechanisms for this class of computations. Also, using OpenGL/Cg rather than CUDA allows using older-generation GPUs (e.g. G7). Our current implementation of SGM with HMI is based on this proven technique, but is also intended as a reference for a future migration to CUDA, which will overcome the OpenGL/Cg limitations described in Section 1.

Our implementation is based on the usage of three render buffers of a frame buffer object. These RGBA-buffers with a 16-bit-float data type (i.e. 1 sign bit, 5 exponent bits, and 10 mantissa bits) are used in a ping pong technique for keeping the data while the arithmetic operations are done in 32-bit-float precision. Almost all work is carried out through the execution of OpenGL rendering commands and several specialized fragment shader programs. For the MI matching table calculation a vertex shader program is used in addition.

4.1 GPU Implementation of SGM

The priority objective for designing the memory buffer partitioning is to provide the input data in a form that minimizes the number of memory accesses in all computations. Generally it is worth keeping in mind that GPUs are principally designed and optimized for fast 3D rendering of textured objects. Calculations should be done on four values synchronously with respect to the super-scalar architecture of the G7 (and earlier) GPUs and the associated data structures in OpenGL/Cg.

Initially, the left and right original images I_L and I_R are considered as textures and loaded with all necessary levels of detail to one color channel of a render buffer. A second channel holds 180 degree rotated versions of the images. According to Section 3.2, a cost volume S is calculated. S is mapped to a render buffer as a sequence of $width/2$ rectangles, each of dimension $disparityrange \times height/2$. Every color channel

of S contains cost values belonging to a quarter of the original image I_L. S stays resident in the render buffer until all path costs for all image pixels have been calculated.

The path costs for one pixel in one path direction are dependent only on the path costs of the predecessor pixel in this direction, not on the costs of the neighboring pixels perpendicular to the path. Therefore all path costs e.g. in horizontal direction for an entire image column can be calculated in parallel. The calculation of path costs for vertical or diagonal directions r is done analogously, respecting the corresponding predecessor dependencies. The computation of four path directions in the four color channels in parallel produces all L_r values by rendering of $width$ rectangles of size $disparityrange \times height$ and $height$ rectangles of size $disparityrange \times width$. Experiments showed that the most efficient way to get the values of the cost volume C for the L_r calculation is to calculate them on the fly rather than storing a separate three-dimensional array C. The values of the adaptive penalty P_2 can either be determined during the calculation of L_r or be pre-calculated and stored on a render buffer. The latter option is slightly faster in practice.

All L_r values depend not only on their predecessor values, according to (7), but also on the minima of the path costs for all disparities for the previous pixel in the current path. Prior to the calculation for the next pixel row and column, some of the L_r channels are shifted in order to optimize the texture fetch in the next stage. For finding the minima of the L_r values for all disparities, a composite procedure of some comparisons in the fragment shader and application of OpenGL blending equation GL_MIN turned out to be the fastest. When the path costs for one column and one row are available they are added to the cost volume S. While the L_r values for one image column can be added by rendering only one rectangle, the values for an image row have to be assigned to S in lines. Care must be taken w.r.t. the range of the values of the L_r because they are accumulated into S through the GPU blending unit with the parameter GL_FUNC_ADD for the blending equation, and the sum is always clamped to the range $[0., 1.]$.

The next step is the disparity selection as described in Section 3.3. The SGM algorithm does not only require computing the minimum *values* of the accumulated path cost values in S but also determining the *indices* of these minimal path costs per pixel, which represent the disparity values. Therefore it is not possible to use the GPU blending unit with blending equation GL_MIN. We implemented a reduce method, which finds both, the values and the positions of the minima for all rows of the rectangles representing S by rendering only a few polygons. In the first reduce step explicit indices are generated and stored together with the values. After the last reduce step, the remaining columns of the rectangles of S contain the minimal path costs and the corresponding raw disparity columns for the quarters of I_L. The sub-pixel interpolation can be integrated into the first reduce step with marginal computational effort.

The cost volume S has to be kept for a diagonal search described in 3.4 for deriving a right disparity image. While all subsequent reduce steps are equivalent to the corresponding ones of the left disparity image, the first reduce pass uses a special address function in the fragment shader. This function handles mapping of the search diagonals to the memory structure of S as described above. In order to work around unexpected results of floating point modulo operations [25] we use a hand-written modulo function. When both disparity images have been derived, they are filtered by a 3×3 median

and the thresholding as described in Section 3.3 is applied for obtaining the final valid disparity image.

4.2 Implementation of the MI Cost Table on the GPU

In order to avoid unnecessary data transfer to and from the CPU, the MI matching table *mi* derived in Section 3.1 is calculated on the GPU, too. Here the main challenge is the calculation of the joint probability distribution P_{I_L,I_R}. This task requires free data scattering and is not possible on the fragment shader level under OpenGL/Cg. However, if all necessary features of OpenGL 2.1 are supported by the GPU a vertex shader program is able to calculate P_{I_L,I_R}[5]. For larger input images, the limited accuracy of the 16-bit float memory buffers available on the GPU requires data partitioning. Thus, the distribution calculation needs to be done for smaller image tiles with a subsequent accumulation phase analogous to [26].

When the joint distribution is available, the single probability distributions are calculated by one-dimensional reduce operations. The unscaled matching table is calculated by applying 5×5 Gaussian filters, logarithm functions and summing in the way described in Section 3.1. Finally, scale factors for that 256×256- array are determined by a two-dimensional reduce operation. They are utilized for adapting the penalties P_1 and P_2 and for setting up the final matching table *mi*.

4.3 GPU Implementation of Hierarchical Computation

The computation of disparity images starts on a coarse level of detail with a random initial matching table. The SGM algorithm is executed and, as described in Section 3.5, the disparity image that is found in this step is up-scaled and a new matching table for the next level of detail is calculated. This iteration terminates when a final disparity image for the original image resolution has been reached. With a G7 GPU, all computationally demanding calculations are done on the GPU, except for the determination of the joint distribution. If GPU and driver support all required OpenGL features (G8 and higher, Section 4.2) all computationally demanding calculation steps from the original stereo image pair to the resulting disparity image are done on the GPU, without hampering the computation by costly data transfers between CPU and GPU.

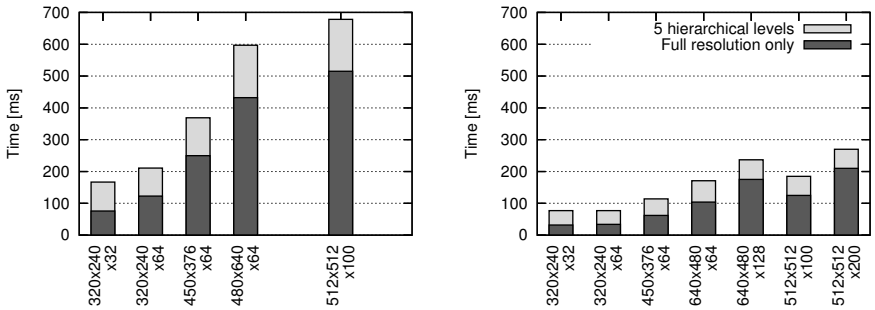
5 Results

We have tested the implementation on a NVIDIA GeForce 7800 GTX with 256 MB as well as a GeForce 8800 ULTRA with 768 MB on-board memory. Disparity images that were computed on the graphics card are shown in Figure 2.

The interesting aspect of the GPU implementation is the run time on different image sizes and disparity ranges. Figure 3 shows the results, which include transfering the stereo image pair onto the graphics card and the disparity image back to main memory.

[5] Unfortunately on the G7 architecture, vertex textures of our render buffer format are not supported. As a workaround the right image can be remapped according to the disparity image on the GPU and after uploading this, the distribution table is computed on the CPU with a slight performance drop.

Fig. 2. Results of computing SGM on the GPU with mutual information and 5 hierarchical levels



(a) GeForce 7800 GTX (b) GeForce 8800 ULTRA

Fig. 3. Run-times of the GPU implementation in different configurations, including the time for transfering the stereo image pair onto the graphics card and the disparity image back to main memory. Not all combinations are possible on the GeForce 7800, due to less memory

In theory, the run-time of the method should scale linear with the number of pixels in an image as well as the disparity range. However, we have found that the runtime on small images (i.e. 320×240) increases only slightly when doubling the disparity range, while the increase comes closer to the expectation when using large images. This is probably because the massive parallelization of the graphics card cannot be properly used for small images. It is similar to the image sizes. The run-time appears to scale linear to the image width and not to the number of pixel of the input images.

In contrast, the run-time scales worse than expected to the number of hierarchical levels. The run-time increase for matching the images at five hierarchical levels is expected to be 14% compared to only matching at full resolution [17]. In our measurements (Fig. 3) it even doubles, on lower image sizes. This is probably due to the constant overhead of MI computation. Fortunately, the relative overhead of MI computation is reduced on larger images.

The original CPU implementation on an Opteron with 2.2 GHz required 1.8 s on images of size 450×375 pixels with 64 pixel disparity range [17]. In contrast, our implementation requires just 114 ms at the same image resolution and disparity range. This is 15 times faster than the CPU implementation.

In comparison, Rosenberg et al. [18] implemented SGM in Cg on a NVIDIA GeForce 7900 GTX. They implemented the absolute difference as matching cost instead of Mutual Information, which is faster, but offers a lower quality. Like us, they computed 8 paths for aggregation (but probably without adapting P_2) and also computed the right image for consistency checking by searching diagonally through the aggregated costs S. In contrast to us, they also did hole filling, but no sub-pixel interpolation. They reached 8 fps using an image size of 320×240 pixel and 64 pixel disparity range. Our full implementation on the probably comparable NVIDIA GeForce 7800 GTX reaches 4.7 fps with the same resolution and disparity range, but includes computing 5 hierarchical levels with mutual information in contrast to Rosenberg et al. With just one hierarchical level, our implementation reaches 8.1 fps. Thus, our implementation has the same efficiency, but allows to compute the full method at the cost of a higher run-time.

Gibson and Marques [19] implemented SGM in CUDA on a NVIDIA Quadro FX5600. They used the more sophisticated sampling insensitive absolute difference [20] instead of Mutual Information. Like us, they implemented 8 paths with adaptive P_2. However, they did not implement consistency checking or sub-pixel interpolation. They reached 5.9 fps using an image size of 450×375 pixel with 64 pixel disparity range. Our full implementation on the probably comparable NVIDIA GeForce 8800 ULTRA reaches 8.8 fps with the same resolution and disparity range, but includes consistency checking and computing 5 hierarchical levels with mutual information in contrast to Gibson and Marques. With just one hierarchical level, but still with the consistency checking, our implementation reaches 16.1 fps. Thus, our Cg implementation appears much more efficient.

6 Conclusion

We have shown that it is possible to implement the full SGM algorithm including pixelwise matching with mutual information on the GPU. Our implementation reaches 4.2 fps on a GeForce 8800 ULTRA with images of 640×480 pixel size and 128 pixel disparity range and 13 fps on images of 320×240 pixel size and 64 pixel disparity range. This is already enough for many real-time applications.

According to reported run-times, our implementation has a comparable efficiency to another Cg implementation of SGM and appeared much more efficient than a CUDA implementation. However, since CUDA offers more flexibility and higher abstraction from the graphics hardware, we are going to implement SGM in CUDA and hope to reach at least the same performance as in our Cg implementation.

References

1. Videre Design: Stereo on chip. http://www.videredesign.com/vision/stoc.htm (2008)
2. Tyzx: Deep sea g2 vision system. http://www.tyzx.com/products/DeepSeaG2.html (2008)

3. Point Grey: Triclops SDK. http://www.ptgrey.com/products/triclopsSDK/index.asp (2008)
4. Hirschmüller, H., Innocent, P.R., Garibaldi, J.M.: Real-time correlation-based stereo vision with reduced border errors. International Journal of Computer Vision **47** (2002) 229–246
5. Owens, J.: GPU architecture overview. In: International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2007 courses, San Diego, CA, USA, ACM (2007)
6. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. In: SIGGRAPH Conference. (2004)
7. NVIDIA: CUDA compute unified device architecture, prog. guide, version 1.1 (2007)
8. Wang, L., Gong, M., Gong, M., Yang, R.: How far can we go with local optimization in real-time stereo matching. In: Third International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT). (2006)
9. Yang, R., Pollefeys, M.: Real-time stereo on commodity graphics hardware. In: IEEE Conference for Computer Vision and Pattern Recognition. (2003)
10. Woetzel, J., Koch, R.: Real-time multi-stereo depth estimation on GPU with approximative discontinuity handling. In March, ed.: 1st European Conference on Visual Media Production, London, UK (2004)
11. Gallup, D., Frahm, J.M., Mordohai, P., Yang, Q., Pollefeys, M.: Real-time plane-sweeping stereo with multiple sweeping directions. In: IEEE Computer Vision and Pattern Recognition, Minneapolis, MN, USA (2007)
12. Cornelis, N., Van Gool, L.: Real-time connectivity constrained depth map computation using programmable graphics hardware. In: IEEE Conference on Computer Vision and Pattern Recognition. Volume 1., San Diego, CA, USA (2005) 1099–1104
13. Yang, R., Welch, G., Bishop, G.: Real-time consensus-based scene reconstruction using commodity graphics hardware. In: Pacific Graphics 2002, Beijing, China (2002)
14. Scharstein, D., Szeliski, R.: Middlebury stereo website. www.middlebury.edu/stereo (2008)
15. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International Journal of Computer Vision **47** (2002) 7–42
16. Gong, M., Yang, Y.H.: Near real-time reliable stereo matching using programmable graphics hardware. In: IEEE Conference on Computer Vision and Pattern Recognition. Volume 1., San Diego, CA, USA (2005) 924–931
17. Hirschmüller, H.: Stereo processing by semi-global matching and mutual information. IEEE Transactions on Pattern Analysis and Machine Intelligence **30** (2008) 328–341
18. Rosenberg, I.D., Davidson, P.L., Muller, C.M.R., Han, J.Y.: Real-time stereo vision using semi-global matching on programmable graphics hardware. In: International Conference on Computer Graphics and Interactive Techniques - SIGGRAPH. (2006)
19. Gibson, J., Marques, O.: Stereo depth with a unified architecture GPU. In: IEEE Conference on Computer Vision and Pattern Recognition. (2008)
20. Birchfield, S., Tomasi, C.: A pixel dissimilarity measure that is insensitive to image sampling. IEEE Transactions on Pattern Analysis and Machine Intelligence **20** (1998) 401–406
21. Hirschmüller, H., Scharstein, D.: Evaluation of cost functions for stereo matching. In: IEEE Conference on Computer Vision and Pattern Recognition, Minneapolis, USA (2007)
22. Kim, J., Kolmogorov, V., Zabih, R.: Visual correspondence using energy minimization and mutual information. In: International Conference on Computer Vision. (2003)
23. OpenGL: Home page. http://www.opengl.org/ (2008)
24. NVIDIA: Cg Toolkit, User's manual, Release 1.4, A Developer's Guide to Programmable Graphics. (2005)
25. Dencker, K.: Cloth Modelling on the GPU. PhD thesis, Department of Computer and Information Science (2006)
26. Scheuermann, T., Hensley, J.: Efficient histogram generation using scattering on GPUs. In: Proceedings of the 2007 symposium on Interactive 3D graphics and games, Seattle, Washington, USA, ACM (2007) 33–37