COMPUTATIONAL OPTIMIZED 3D RECONSTRUCTION SYSTEM FOR AIRBORNE IMAGE SEQUENCES

K. Zhu[1], M. Butenuth[1] & P. d´Angelo[2]

[1]Technische Universität München, Remote Sensing Technology, D-80333 München, Germany,
{ke.zhu, matthias.butenuth}@bv.tum.de
[2]German Aerospace Center (DLR), Remote Sensing Technology Institute, D-82234 Wessling, Germany
pablo.angelo@dlr.de

Commission I

KEY WORDS: Near Real-time 3D Reconstruction, Semi-Global Matching (SGM), Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA)

ABSTRACT:

In this paper, a computational optimized 3D reconstruction system for airborne image sequences is described and tested. The onboard system is designed to generate automatically and continuously the Digital Elevation Model (DEM) for large observation areas during the flight. The aim is to provide near real-time applications such as supporting rescue teams for prompt evaluation and timely reactions after natural disasters such as floodings, landslides or earthquakes. In addition, the proposed system enables the fast and accurate processing of 3D reconstruction for urban areas. The different optimization aspects and software-hardware methods are combined in the system as a compact solution to squeeze the potential performance of devices and to shorten the calculation time. Currently, most real-time stereo systems are applied only to small image sizes deriving a fast processing. Implementations on Graphics Processing Units (GPUs) are posed as image rendering. In this paper, a computational orientated concept for stereo processing using large images without special hardware is introduced. The data is partitioned according to the memory architecture of the programmable GPUs by parallel processing. This optimization is implemented on GPU with Compute Unified Device Architecture (CUDA). First results demonstrate the potential of the proposed real-time strategy.

1. INSTRODUCTION

In this paper, a computational optimized 3D reconstruction system using programmable Graphics Processing Units (GPU) is described and evaluated. Real-time stereo reconstruction systems are much in demand for many applications including support after natural disasters such as floodings, landslides or earthquakes, the derivation of a fast 3D reconstruction of urban areas and close range applications. Commercial Central Processing Unit (CPU) implementation is unable to perform the generation of Digital Elevation Model (DEM) at a continuously frame-rate for large observation areas. Since last year modern GPUs permeate steadily in scientific and engineering fields by means of their specialized design for compute-intensive, massively data parallel computation. In 2007, NVIDIA introduced a new and flexible concept, the Compute Unified Device Architecture (CUDA) (NVIDIA, 2007) that enables the General-Purpose Computation on Graphics Hardware (GPGPU) in the familiar C programming language. It affords directly management on memories with different characteristics. Thus, it is no more necessary to disguise algorithms such image rendering.

In the proposed system, the input images are firstly rectified according to a compact rectification algorithm (Fusiello & Trucco, 2000). The important advantage of rectification is that computing stereo correspondences is done along the horizontal or vertical lines of the rectified images. It simplifies the index searching and leads a sequential movement in memory. Currently, almost all implementations for real-time accurate dense 3D reconstruction are based on global stereo approaches. They are more accurate than local methods at the cost of their computation complexity (Scharstein & Szeliski, 2002). The Semi-Global Matching (SGM) (Hirschmüller, 2009) is selected for the stereo algorithm, because the Middlebury online evaluation (Scharstein & Szelisk, 2010) demonstrates that it is one of the currently best global stereo methods. However, providing accurate results, the run-time of SGM for large images is still the bottleneck for 3D reconstruction processing. On the other site, its parallel execution model accords with Single Instruction Multiple Threads (SIMT) architecture. Thus, the proposed efficient processing in this paper is based on a CUDA implementation. The novel core of the approach is to optimize the SGM method for large images considering a suitable data tiling on GPU.

The implementation of the SGM is so far realized on GPUs with OpenGL/C for Graphics (Cg) which utilizes older-generation GPUs having fewer capabilities and less programmability (Ernst & Hirschmüller, 2008). In contrast, CUDA is much less constrained by general purpose computation. In addition, the real-time processing above was reached 4.2 fps on GeForce 8800 ULTRA for images having 640×480 pixels of 128 pixels disparity range.. In the work of (Hirschmüller, 2008) large image sizes are used, but the matching of one image (86 MPixel) against six neighbors took around 5.5 hours on one 2.2 GHz Opteron CPU. In the paper of Gibson and Marques (Gibson & Marques, 2008) the execution time for cost aggregation in their CUDA Implementation is not explicitly declared. Besides, the run-time for 450×375 images with 64 disparity depth are even worse as the OpenGL approach reaching an efficient of 3 times better than their CPU implementation.

The goal of this paper is to parallelize the massive computation parts on GPU, to optimize the data transformation in the processing steps and between them and, finally, to reach a near real-time stereo processing. The presented work refers mainly to three parts: in the next Section 2, the used methods are described. In Section 3, the design idea of parallelization using modern GPUs is illustrated. In Section 4, first experimental results of the realized implementation are analyzed. Finally, further work is discussed in Section 5.

2. 3D RECONSTRUCTION USING SEMI-GLOBAL MATCHING

2.1 Strategy for 3D reconstruction

In this section, the methodical steps of the stereo processing are demonstrated, shown as flow diagram in Figure 1. The input images using Global Positioning System (GPS) data and camera parameters are corrected and rectified to epipolar images, which enable a linear storage in memory. Afterwards, a correspondence method, the Semi-Global Matching, is used to generate disparity images. Generally, the Semi-Global Matching method consists of four steps like most methods in global category: matching cost computation, cost aggregation, disparity computation and disparity refinement.

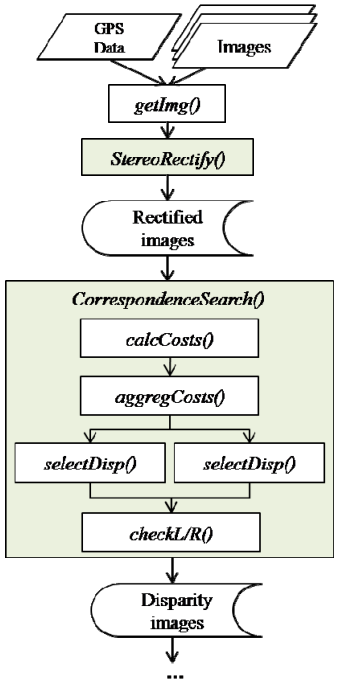

Figure 1: Flow diagram for stereo 3D reconstruction

2.2 Rectification of stereo pairs

The perspective transformation for homogeneous coordinates of object points $\vec{X}$ into the image coordinate system can be presented with the following equation:

$$ {}^{H}\vec{x} = KR^{T}\left[\vec{I}_{3\times3} \quad -\vec{X}_{0}\right] \times {}^{H}\vec{X} = Q\left[\vec{I}_{3\times3} \quad -\vec{X}_{0}\right] \times {}^{H}\vec{X} \qquad (1) $$

The vectors ${}^{H}\vec{x}$ and ${}^{H}\vec{X}$ present the image and object points in the homogeneous coordinate system, respectively. $\vec{X}_0$ is the camera centre. The Q matrix summarizes the rotation matrix R and the camera matrix K. The idea of rectification is to define two new perspective projection matrices obtained by rotating the old ones around their optical centres until focal planes becomes coplanar containing the baseline. The new projection matrices have the same rotation matrix:

$$ \begin{aligned} P_{nL} &= K_{nL}R_{n}^{T}\left[\vec{I}_{3\times3} \quad -\vec{X}_{0L}\right] \\ P_{nR} &= K_{nR}R_{n}^{T}\left[\vec{I}_{3\times3} \quad -\vec{X}_{0R}\right] \end{aligned} \qquad (2) $$

P_{nL} and P_{nR} are the new projection matrices for the left and right epipolar images. They vary in the principal point in x-direction included in K_{nL} and K_{nR} and their projection centres between $\vec{X}_{0L}$ and $\vec{X}_{0R}$. According to epipolar geometry, the pixels in the original image and in the new generated epipolar image have the following relation:

$$ \begin{aligned} \vec{X} - \vec{X}_{0} &= Q_{L}^{-1\,H}\vec{x}_{L} = (K_{L}R_{L}^{T})^{-1\,H}\vec{x}_{L} \\ \vec{X} - \vec{X}_{0} &= Q_{nL}^{-1\,H}\vec{x}_{nL} = (K_{nL}R_{n}^{T})^{-1\,H}\vec{x}_{nL} \end{aligned} \qquad (3) $$

It is essential that:

$$ \begin{aligned} {}^{H}\vec{x}_{nL} &= T_{L}\,{}^{H}\vec{x}_{L} \\ T_{L} &= Q_{nL}Q_{L}^{-1} = (K_{nL}R_{n}^{T})\,(K_{L}R_{L}^{T})^{-1} \end{aligned} \qquad (4) $$

In a similar way, the transformation matrix can be derived for the right image. Via this pair of rectifying projection matrices the conjugate epipolar lines become collinear and parallel to one of the image axes. This preprocessing simplifies the subsequent dense stereo matching.

2.3 Correspondence research using Semi-Global Matching

Two factors are considered when choosing a method for correspondence searching on GPU. First, global matching methods are more accurate than local methods (Scharstein & Szeliski, 2002), but their run-time makes them unsuitable for real-time applications. As an exception, the cost complexity of the Semi-Global Matching is *O(width×height×disparity_range)* like local methods (Hirschmüller, 2008). Second, its methodical realization has a regular structure and maps the SIMT mechanism of GPUs (NVIDIA, 2009a). Thus, an efficient computation using programmable GPUs is possible.

The matching cost for two pixels can be derived from different methods. The absolute differences between pixel intensities are used as correspondence cost. An extension with i.e. Mutual Information (MI) is suitable which performs a better matching (Hirschmüller & Scharstein, 2007). In the disparity range belong epipolar lines the gray values are read and pointed into the cost table, which is generated from the histogram of intensities of both images:

$$ C(p, d) = L(p) - R(p + d) \qquad (6) $$

where C = cost for pixel for p at d
d = absolute disparity from p
$L(p)$ = intensity of pixel p in the left image
$R(p+d)$ = intensity of pixel $(p+d)$ in the right image.

An example of this step is visualized in Figure 2: *u* and *v* are the intensities in the left and the right image, *c* is the cost from the table on the right size.

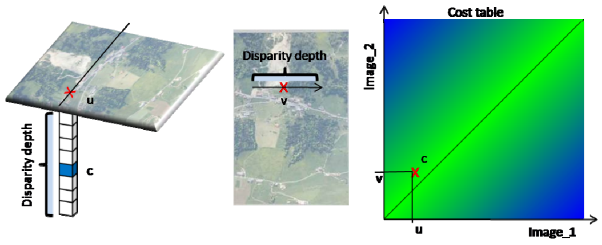

Figure 2: Cost generation from lookup table

The SGM method approximates the minimization of the global energy $E(D)$:

$$E(D) = \sum_{p} (C(p, D_p) + \sum_{q \in N_p} P_1 T\left[\left|D_p - D_q\right| = 1\right] + \sum_{q \in N_p} P_2 T\left[\left|D_p - D_q\right| > 1\right] \quad (7)$$

The first term sums the costs of all pixels in the image with their particular disparities D_p. The next two terms penalize the discontinuities with penalty factors P_1 and P_2, which differ in small or large disparity difference within a neighbourhood q of the pixel p. This minimization approximation is realized by aggregating $S(p, d)$ of path wise costs into a cost volume:

$$S(p, d) = \sum_{r} L_r(p, d) \quad (8)$$

$L_r(p, d)$ in (8) represents the cost of pixel p with disparity d along one direction *r*. It is described as following:

$$\begin{aligned} L_r(p, d) = {} & C(p, d) + \\ & \min\{L_r(p - r, d), \\ & L_r(p - r, d - 1) + P_1, \\ & L_r(p - r, d + 1) + P_1, \\ & \min_i L_r(p - r, i) + P_2\} - \\ & \min_i L_r(p - r, i) \end{aligned} \quad (9)$$

This smoothed cost function considers the change continuity in a direction of cost aggregation as well as non identical disparities of the local pixel with prior pixel. The disparity at each pixel is selected as index of the minimum cost from the cost cube.

3. COMPUTATIONAL OPTIMIZATION ON GPU

3.1 Strategy of real-time stereo processing

From the hardware viewpoint, the GPUs consist of a set of Streaming Multiprocessors (SM), each of them computing several warps of threads. Warp is the unit definition about a group of 32 threads, which is the minimum data processing size in SIMT fashion in SM (NVIDIA, 2010). All threads run the same codes on GPU and communicate with each other via shared memory. The basic idea of the aimed strategy is the partitioning of the related data, tiling them from global memory on device into the private shared memory of the multiprocessors, performing each thread with localized data and, afterwards, copying the results back to global memory.

Figure 3 visualises the data flow diagram in different memory levels of CPU and GPU during stereo processing. Firstly, the input images are loaded from CPU memory to the global memory on GPU. The remaining processing stays in GPU until copying the results back. Important is here, the available data resides in GPU to avoid the lagged data transformation between host and device via PCI-Express bus (Nukada et al. 2008). Secondly, in the cost calculation step, the rectified images are stored in global memory as texture and tiled line-by-line into shared memory in SM, whose latency is roughly 100× lower than global memory latency (NVIDIA, 2009b). Thirdly, each data element in the cost cube maps a thread in GPU and is pathwise aggregated.

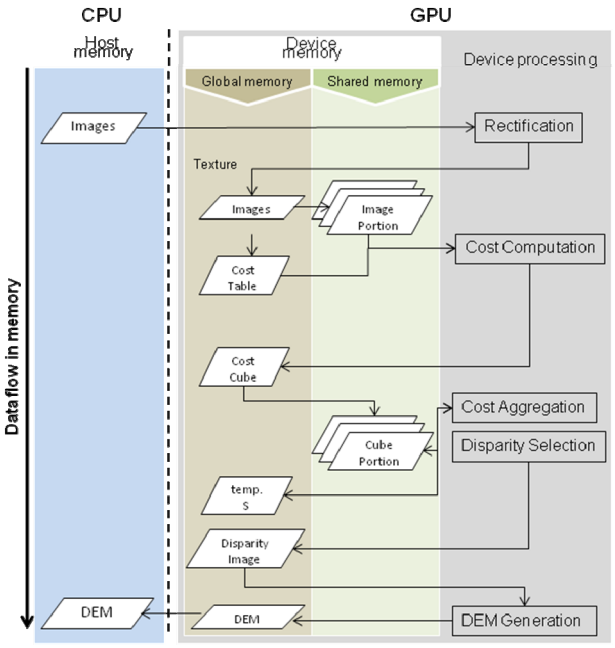

Figure 3: Data flow in CPU and GPU memories during 3D reconstruction

3.2 Rectification on GPU

The parallelization of the rectification step requires no iterative data use. The values of all pixels from the original image are tiled in blocks. The image index is mapped from the combination of the block ID and thread ID according to their declaration. Two transformation matrices T_L and T_R must be defined locally in the kernel-function in order to use the quick caching of broadcast mechanism. It means, the required data will be read only once and propagated for all threads. The pixels are rectified concurrently with the matrix using the Equation 4. Their intensities are rewritten with the new coordinates in the rectified image, which is allocated in the global memory.

3.3 Matching cost computation

The costs in the cost lookup table are simply identical with the absolute intensity difference. This table can be refined with

hierarchical mutual information (Hirschmüller, 2008). In the cost computation step, each pixel in the left image is compared with all reference pixels in the disparity range from the right image. The accordant matching costs are read from the cost table using their intensities as indices. In fact, a pixel from the left image is related with all pixels between minimum and maximum disparity in the right image. Storing of image intensities in global memory induces large memory accesses: for cost calculation on one pixel, dis_{range} times private readings is required, however the coming pixel could share $dis_{range} - 1$ already read data with the previous pixel. The values in the image are partitioned line-by-line and tiled into the shared memory to reduce the memory accesses on global memory and increase the data utilization rate. The ground design ideal is visualized in Figure 4.

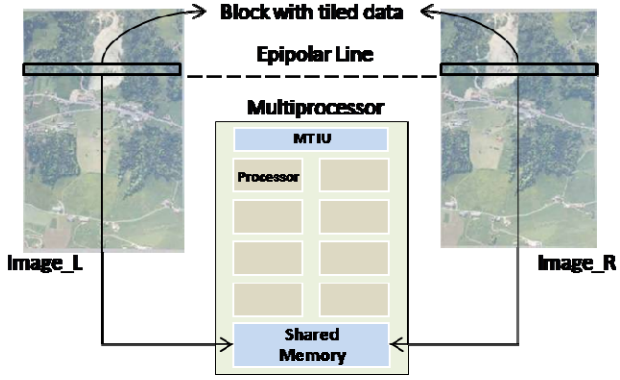

Figure 4: Data tiling in shared memory for cost computation

The available amount of shared memory in a SM is 16KB, which is enough for storing 4000 pixels in `float`. But the maximum number of resident threads per multiprocessor is 1024 for GPUs with compute capability 1.3. Hence the maximal threads number is the decisive factor for parallelization. The block dimension is designed in 2D, because the maximum size of the x-, y-, and z-dimension of a thread block is limited for all GPUs up to Compute Capability 1.3 at 512, 512, and 64, respectively (NVIDIA, 2010). One dimension block for large image exceeds the hardware competence.

In the kernel function a barrier synchronization call ensures that all required data for the next step is already updated to the shared memory before their individual calculations. Otherwise, threads could get empty values from uninitialized vectors. Figure 5 illustrates a block processing for generating a cost wall from the tiled data.

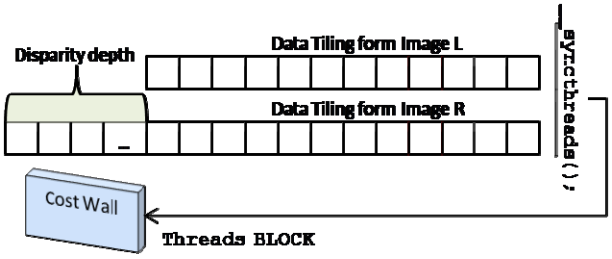

Figure 5: Threads allocation and synchronization by cost wall generation

Consequently, each thread in a block answers to a pixel in the image line. A threads block generates a part of the complete cost cube. The excerpt of the `CostCal` reports the CUDA kernel code:

```
__global__ void CostCal (…){
   const int ix = threadIdx.x;
   const int iy = blockIdx.x;
   …
   __shared__ float1 sData_l[IMG_LENGTH],
                     sData_r[IMG_LENGTH];

   //Calculation over complete image
   for (int l = 0; l<imgH; l++){

     //Update intensities from texture
     sData_l[ix] = tex2D(l_texImg, ix, l);
     sData_r[ix] = tex2D(r_texImg, ix, l);
     __syncthreads();
     …
     //Update intensities from texture
     for(int DStep = 0; DStep<DDepth; DStep++){
          //Calculation
         cost = tex2D(ct_texImg, ix, iy);
          d_ccube[DStep*imgW*imgH + imgW*l + ix]
           = cTable(sData_l[ix].x,sData_r[ix].x);
     }
   }
}
```

3.4 Cost Aggregation and Disparity selection

The CPU implementation for cost aggregation is typically a serial computation along different paths. In contrast, the proposed implementation uses the advantage of the threads parallelization of GPUs to calculate more data elements synchronously. The cost optimization for each pixel in one direction requires two mass groups of data: the own lookup costs in the disparity range, which are generated in the past section, and the optimized costs from the previous pixel in the path. A pixel in the cost cube contains dis_{range} data elements that map the same amount threads. Moreover, the massive accessing on same data suits exactly the advantage of shared memory on GPUs. The previous optimized costs are repeatedly used for all data elements in the next pixel. The memory requirement for this step is potentially to the data quantity, which has twice over dis_{range} elements for every pixel.

A further challenge is that pixels are no more independently with each other like in the cost computation. They take the optimized results backward along a path. Instead of rectangular data tiling from image matrix, a line scanning of thread allocation is swept through the image. This ground ideal for parallelization is visualised in Figure 6.

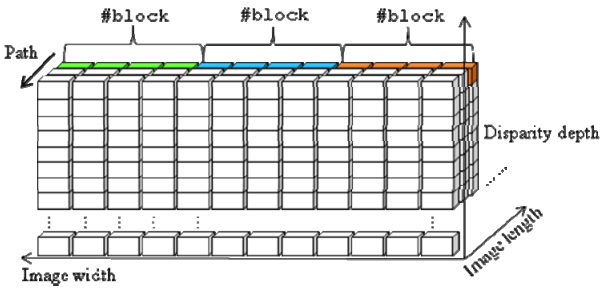

Figure 6: Parallelization of cost aggregation

Traditionally, the sweeping should be executed more times for e.g. eight directions SGM. The quick implementation tries just two passes, from top to bottom and from bottom to top, to achieve the cost optimisation in six directions. The disparity selection is done after the cost aggregation in the second sweeping. It is simply to extend for more directions, if the sweepings start from another sides of the image. This quick approach is shown in Figure 7. The border pixels receive their $C(p, d)$ as $S(p, d)$. Each element in the cost cube maps to a thread. In a tiling block $dis_{range} \times BLOCKSIZE$ threads are concurrently in cost optimizing according to the Equation 9.

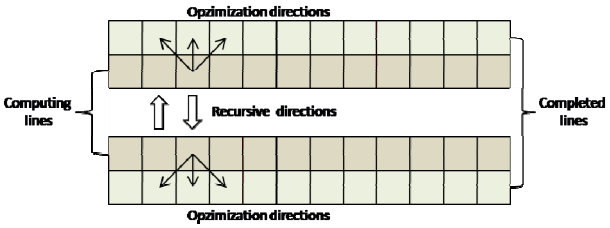

Figure 7: Fast processing using two sweeping through complete image

Furthermore, the meanwhile optimized cost wall is written directly to global memory, but not eliminated from shared memory, because they can be used for the next line. This finesse avoids reading data from optimized cost and economizes the expensive memory accessing to global memory for each block. The data flow during kernel execution is presented in Figure 8. This strategy can be used only for the up-down and down-up directions in the quick approach. The skewing in the other directions requires a global view of optimized costs, in order to enable the communication between blocks.

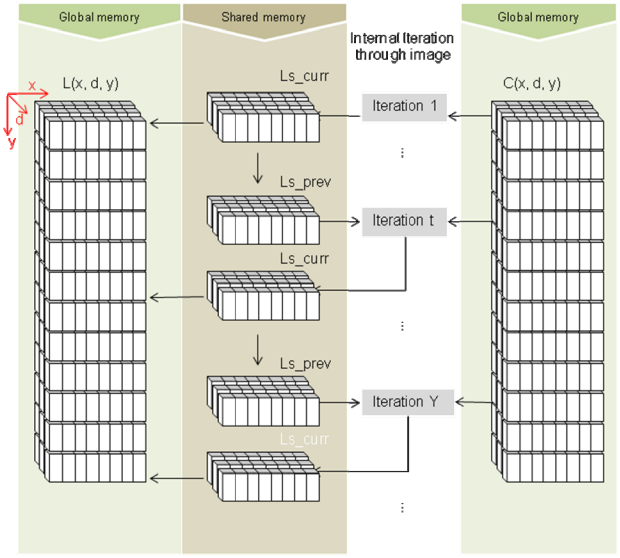

Figure 8: Data flow during cost aggregation (one direction)

4. EXPERIMENTAL RESULTS

The experiment results are computed on a NVIDIA GeForce GTX 295 graphics card. One of the both GT200 graphic processor is used for the calculation. This device core has 30 Streaming Multiprocessors on-chip and suffices 1.3 CUDA compute capability (NVIDIA, 2010). The compared CPU implementation runs on an Intel Core2 Q9450 CPU with 6 MB L2 Cache. The resulted disparity image of GPU implementation is shown in Figure 10, one of the related input images in Figure 9. The pair of arranged stereo images has a pixel size of 1000×1000 with a disparity range of 80.

The un-optimized CPU implementation needs about 5200 ms to finish the stereo processing including rectification. In contrast, the GPU-accelerated implementation requires just 722 ms totally. A comparison of the execution time between CPU and GPU implementations with the above referred example are presented step by step in Table 1. In each step the improved version on GPU is more efficient than un-optimized CPU execution. By the critical part, cost aggregation, the GPU implementation takes an excellent performance of about 9 times faster than the CPU processing. Finally, the GPU improvement

Figure 9: Left input image, Munich Frauenkirche

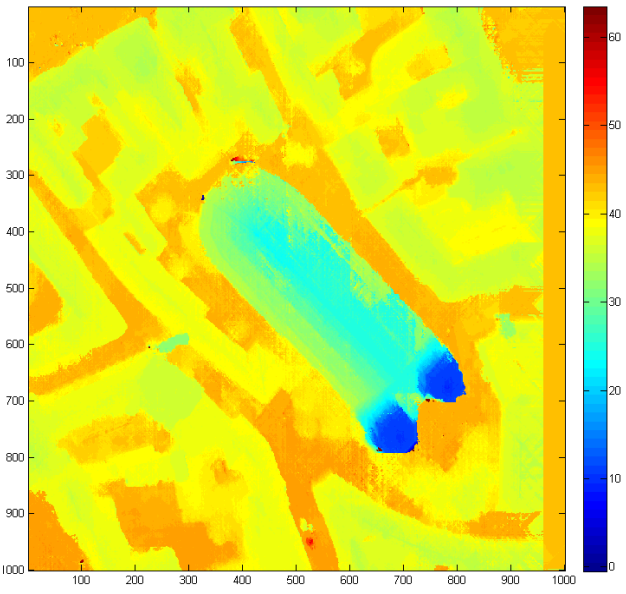

Figure 10: Resultant disparity image of the GPU implementation using CUDA

	CPU Implementation	GPU Implementation
Rectification	432 ms	96 ms
Cost calculation	200 ms	9 ms
Cost aggregation	4215 ms	481 ms
Disparity selection	362 ms	136 ms
Total	5209 ms	722 ms

Table 1: Comparison of execution times

using CUDA reaches at least seven times as the commercial implementation on CPU. By the last step in the table, disparity selection, its execution time can be reduced, if an appropriate block size is used. This part of implementation uses the identical block tiling size in this paper in order to gain the cost aggregations from left to right and right to left.

The accuracies between CPU and GPU implementation are compared. Generally, they take similar results as shown in Figure 11. In this scenario, the disparities on the church roof of the GPU result are better than CPU execution, in respect that the left to right and right to left cost aggregations on GPU use a different P_2 with other aggregation paths. The used intensity based matching cost is very sensitive to illumination differences, reflections, etc. Alternatively, mutual information registers complex radiometric relationships and can bring better results (Hirschmüller, 2009).

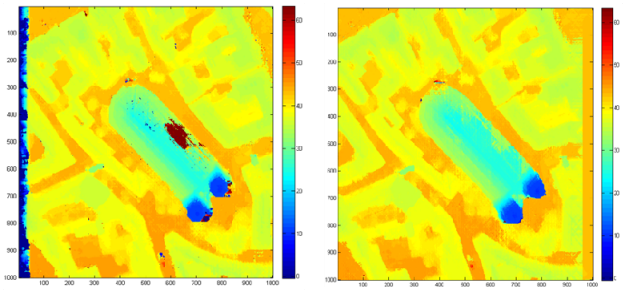

Figure 11: Result comparison between CPU (left) and GPU (right) implementation

5. CONCLUSIONS

The proposed work indicates large potential capability of common graphics cards for general computation. The computational improvement on programmable GPUs stereo processing has shown that it is possible to obtain near real-time 3D reconstruction without special hardware such as Field Programmable Gate Array (FPGA) (Gehrig et al., 2009). The parallelization using CUDA is not necessary to pretend as image rendering using Cg. This enables the developers more flexibilities on programming and requires no additional software skills about 2D/3D Application Programming Interface (APIs).

For the first implementation, the absolute intensity differences are used as costs in the lookup table. Future work will contain the mutual information. In addition, a full SGM algorithm with left and right disparities check will be implemented. The pathwise aggregation of SGM is still an obstacle for data parallelization, because each template summery $S(p, d)$ of optimized costs $L_r(p, d)$ must be global viewable for all blocks. This property caused memory access latency cannot be avoided. In future work, different tiling strategies will be achieved and compared. A more efficient and accurate stereo processing on GPU is on-going work.

ACKNOWLEDGEMENTS

The images are provided by the 3K camera system of the German Aerospace Center.

REFERENCES

Ernst, I., Hirschmüller, H., 2008: Mutual Information based Semi-Global Stereo Matching on the GPU. In: *4th International Symposium on Visual Computing (ISVC08)*, USA.

Fusiello, A., Trucco, E., Verri, A., 2002. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications* 12(1), pp. 16-22.

Gehrig, S. K., Eberli, F., Meyer, T., 2009. A Real-Time Low-Power Stereo Vision Engine Using Semi-Global Matching. *International Conference on Computer Vision Systems, Lecture Notes in Computer Science* 5815, pp. 134-143.

Gibson, J., Marques, O., 2008: Stereo Depth with a Unified Architecture GPU. In: *Workshop on "Computer Vision con GPUs"*, co-located with *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, USA

Hirschmüller, H., 2008. Stereo processing by semi-global matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30(2), pp. 328-341.

Hirschmüller, H., Scharstein, D., 2007. Evaluation of cost functions for stereo matching. In: *IEEE Conference on Computer Vision and Pattern Recognition, Minneapolis*, USA

Hirschmüller, H., Scharstein, D., 2009. Evaluation of Stereo Matching Costs on Image with Radiometric Differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31(9), pp. 1582-1599.

Nukada, A., Ogata, Y., Endo, T., Matsuoka, S., 2008. Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA. In: *Conference on High Performance Networking and Computing*, USA.

NVIDIA, 2007. NVIDIA CUDA Programming Guide, Version 1.0.

NVIDIA, 2009a. CUDA Architecture, Introduction & Overview, Version 1.1.

NVIDIA, 2009b. OpenCL Best Practices Guide, Version 1.0.

NVIDIA, 2010. NVIDIA CUDA Programming Guide, Version 3.0.

Scharstein, D., Szeliski, R., 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision* 47(1/2/3), pp. 7-42.

Scharstein, D., Szelisk, R., 2010: Middlebury stereo website. www.middlebury.edu/stereo (accessed 15. April 2010).