Comparison of Dense Stereo Using CUDA

Ke Zhu[1], Matthias Butenuth[2], and Pablo d'Angelo[3]

[1] Technische Universität München, Remote Sensing Technology, Germany
ke.zhu@bv.tum.de
[2] IAV GmbH, Active Safety and Driver Assistance, Germany
matthias.butenuth@iav.de
[3] German Aerospace Center (DLR), Remote Sensing Technology Institute, Germany
pablo.angelo@dlr.de

Abstract. In this paper, a local and a global dense stereo matching method, implemented using Compute Unified Device Architecture (CUDA), are presented, analyzed and compared. The purposed work shows the general strategy of the parallelization of matching methods on GPUs and the tradeoff between accuracy and run-time on current GPU hardware. Two representative and widely-used methods, the Sum of Absolute Differences (SAD) method and the Semi-Global Matching (SGM) method, are used and their results are compared using the Middlebury test sets.

1 Introduction

In this paper, two representative and widely-used dense matching methods of stereo processing for near real-time 3D reconstruction using Compute Unified Device Architecture (CUDA) on programmable GPUs are described and evaluated. Real-time stereo reconstruction is a very active research topic in computer vision and is required for many applications such as remote sensing tasks and close range applications. Compared with feature-based methods, the dense matching methods are less sensitive with application scenarios and can be used more diffusely for both video sequences in robotics and large observation images from airborne system [1]. Generally, the taxonomy divides dense stereo matching methods in local (block-based) and global methods. Global stereo methods show the best performance on the Middlebury online evaluation [2] than the simpler local methods without special postprocessing steps.

The Semi-Global Matching (SGM) method is selected, because it is a high performance global stereo method, but retains a complexity that is linear to the reconstructed volume. It is realized for many practical applications like in automobiles on FPGAs and the earth observation tasks on CPU [3]. The local methods are favorited by real-time applications because of their simple and fast implementations. Their mechanisms are similar and can be rephrased or extended from the Sum of Absolute Differences (SAD) method [4].

In 2007, the G80 series graphics card of NVIDIA was introduced based on the CUDA Architecture that enables the General-Purpose Computation on Graphics Hardware (GPGPU) in a familiar C programming language [5]. Compared to the

K.N. Kutulakos (Ed.): ECCV 2010 Workshops, Part II, LNCS 6554, pp. 398–410, 2012.
© Springer-Verlag Berlin Heidelberg 2012

earlier GPGPU programming paradigm, CUDA does not need the reformulation of the algorithms into a computer graphics rendering framework. This eases the implementation and allows more flexible use of the GPU hardware.

The novel core of our approach is to find a combination between the methods and the hardware, to demonstrate the general parallelization strategy of matching methods on GPUs and compare two different dense stereo matching algorithms. The presented work consists of three parts: the next section includes the basic algorithms and introduction to CUDA, section 3 the implementation of them on GPU and section 4 an evaluation using the Middlebury stereo images.

2 Basics

2.1 The Dense Matching Methods

A taxonomy of existing stereo algorithms is provided in [6]. Generally, the stereo algorithms perform four steps: matching cost computation, cost aggregation, disparity optimization and selection and disparity refinement. The taxonomic branch appears in the disparity optimization step: local methods perform a block-based "winner-take-all" optimization at each pixel in the aggregation step. In contrast, global methods skip it and are formulated in an energy-minimization frame work.

Generally, the disparity computation of local algorithms depends only on the intensities within a finite window. The Sum of Absolute Differences (SAD) algorithm is selected as an example, as it can be easily parallelized due to its simple structure. It can be described in the following steps: the cost of pixel *p(r,c)* is the absolute difference of intensity values at the given disparity *d*. The cost aggregation is done by summing of matching costs over the window. Disparity is selected with the minimal aggregated value:

$$S(p(r,c),d) = \frac{1}{4 \times n \times m} \times \sum_{m}^{-m} \sum_{n}^{-n} |I_1(r+i, c+j) - I_2(r+i, c+j+d)|. \quad (1)$$

In contrast, global algorithms perform almost all of their work in the disparity optimization step. The Semi-Global Matching (SGM) method is chosen, because of its accuracy and computational complexity as $O(width \times height \times DisparityRange)$ like local methods [7]. Its methodical realization has a regular structure and maps to the Single Instruction Multiple Data (SIMD) mechanism of GPUs.

The matching cost for two pixels can be derived from different methods. The absolute differences between pixel intensities are used as correspondence cost. For larger baselines other cost functions, such as Mutual Information, result in a better performance [8], but are not evaluated here:

$$C(p,d) = L(p) - R(p+d), \quad (2)$$

where $C(p,d)$ is the cost of pixel p at the disparity d. $L(p)$ and $R(p,d)$ denote the intensities in the left and right image respectively.

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

$$E(D) = \sum_p (C(p, D_p) + \sum_{q \in N_p} P_1 \left[|D_p - D_q| = 1\right] + \sum_{q \in N_p} P_2 \left[|D_p - D_q| > 1\right]). \quad (3)$$

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 differences 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:

$$S(p, d) = \sum_r L_r(p, d). \quad (4)$$

$L_r(p, d)$ in (4) represents the cost of a pixel p with disparity d along one direction r. It is described as following:

$$\begin{aligned} L_r(p, d) = C(p, d) + min\,(L_r(p - r, d), L_r(p - r, d - 1) + P_1, \\ L_r(p - r, d + 1) + P_1, \min_i L_r(p - r, i) + P_2) - \min_i L_r(p - r, i). \end{aligned} \quad (5)$$

This regularization term function favors planar and sloped surfaces, but still allows larger height jumps in the direction of cost aggregation. The disparity at each pixel is selected as the index of the minimum cost from the cost cube.

2.2 Compute Unified Device Architecture (CUDA)

The computational design for Semi-Global Matching concerns not only the parallelization mechanism but also the limitations of the hardware. Hence, in this subsection the basic concepts of CUDA programming as well as the method-involved physical features are introduced.

CUDA divides the computation units into hosts, such as a CPU and device, normally such as a GPU. Massively parallel processing runs on the device during the *kernel*-functions. The kernels specify the code to generate a large number of threads to exploit data parallelism. They are organized in blocks and refer via the thread indices in 1D, 2D or 3D. A Warp is defined as a group of 32 threads, which is the minimum data processing size in a SM (Streaming Multiprocessor). All of these threads within a block execute the same code as well-known Single-Program, Multiple-Data (SPMD) parallel programming style [5]. Threads in the same block share data and synchronize while doing their share of the work. Figure 1 left shows CUDA thread organization. Each SM in graphics cards with compute capability 1.3 can take up to 8 blocks and allows maximal 1024 threads [9].

CUDA enabled devices have separate memory spaces with different characters. Figure 1 right shows an overview of the device memory model. Global and

constant memories are used for data transfer between host and device. The constant memory allows read-only access and allows quick caching using a broadcast mechanism. The shared memory is the key for hidding memory access latency to avoid bandwidth saturation. Its latency is roughly 100 times faster than global memory latency [10]. The profitable strategy for performing computation on GPUs using this advantage is to partition data in subsets, copying them from global memory to shared memory using multiple threads, achieving them from shared memory locally in threads and copying the results back to global memory. The amount of shared memory per SM is 16 KB [9] and must be noticed by design for data tiling.

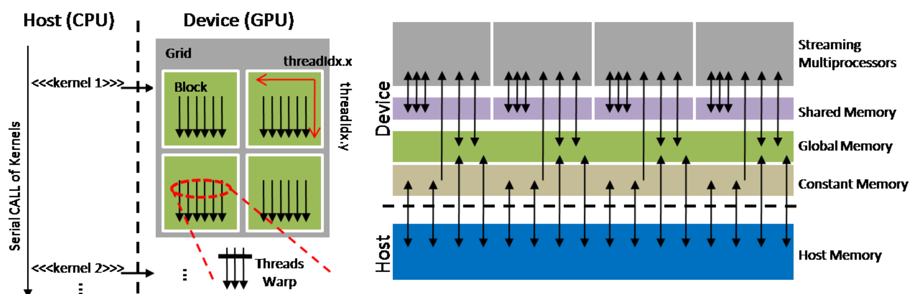

Fig. 1. Overview of the CUDA threads (left) and memory model (right)

3 GPU Implementations Using CUDA

3.1 Strategy for Real-Time Stereo Processing

The preprocessing for unconstrained stereo rigs to simplify the correspondence searching is the rectification of images using a compact algorithm of [11] on GPU. The pixel coordinates are mapped with the combination of the block ID and thread ID. The new pixel coordinates in the epipolar images are generated using locally defined transformation matrices in the kernel. The resulted epipolar images enable a linear correspondence searching. The remaining stereo processing stays in GPU until copying the results back.

The SAD and SGM method use the same GPU implementation by the cost calculation step: the absolute differences for each pixel along an epipolar line are calculated synchronously using the line-by-line tiled data in the shared memory. Like that, the SAD method uses the similar way to parallelize its cost aggregation on GPU. The disparities are selected directly after the aggregation. Differently, by the cost optimization of SGM method each data element in the 3D cost cube maps a thread in GPU and is path wise aggregated into the optimized cost cube in global memory. As disparity refinement, an additional median filter is implemented. The left-right check can be executed using the same processing with exchanged data sequence.

3.2 Matching Cost Calculation

In the cost computation step, each pixel in the left image is compared with all reference pixels in the disparity range of 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. 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, because each pixel from the right image can be used $(DisparityRange - 1)$ times. The ground design idea is visualized in Figure 2.

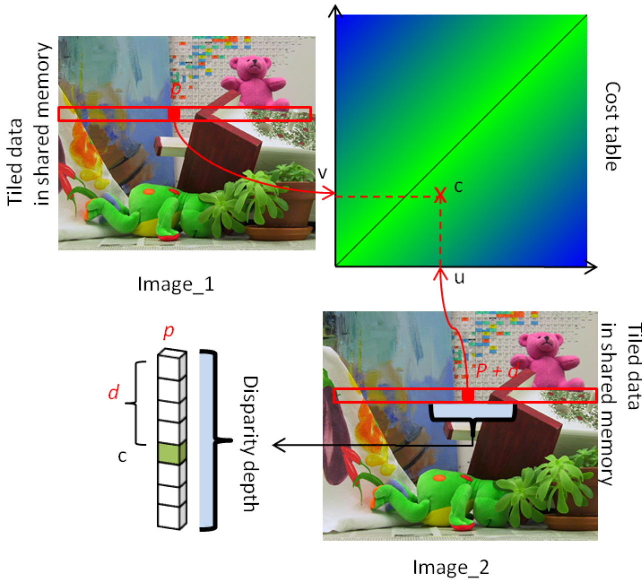

Fig. 2. Generation of cost cube from tiled data using lookup table: u is the intensity of pixel p in the first image, v is the intensity of his corresponded pixel in the second image along the epipolar line, c is the cost reading from the cost table

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 the compute capability 1.3 at 512, 512, and 64, respectively [9]. One dimension block for large images 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. 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(...){
   __shared__ float1 sData_l[IMG_LENGTH],
                     sData_r[IMG_LENGTH];

   for (int l = 0; l<imgH; l++){//over complete image
     //Update intensities from texture
     sData_l[ix] = tex2D(l_texImg, threadIdx.x, l);
     sData_r[ix] = tex2D(r_texImg, threadIdx.x, l);
     __syncthreads();

     for(int DStep = 0; DStep<DDepth; DStep++){
            //Cost Calculation
            cost = tex2D(ct_texImg, threadIdx.x, blockIdx.x);
            d_ccube[DStep*imgW*imgH + imgW*l + threadIdx.x]
             = cTable(sData_l[ix].x,sData_r[threadIdx.x].x);
      }
   }
}
```

This separate cost computation step is only required for the SGM method.

3.3 Cost Aggregation

In the local SAD method, the disparity is computed for all pixels independently. In this case, the cost computation, cost aggregation and disparity selection steps can be computed in parallel, without requiring additional storage space. The main CUDA kernel for the local SAD method is shown below:

```
for (int di = 0; di < Depth; di++){
    sad = 0;
    //Aggregation in windows
    for (int wj = 0; wj < w_width; wj++){
        for (int wi = 0; wi <w_height; wi ++){
            sad = sad + abs(tData[...].x - mData[...)].x);
        }
    }
    if (di == 0) tempM = sad; selI = 0;
    if (sad < tempM) tempM = sad; selI = di;
}
d_disp[imgW*iy + ix] = make_color1(selI);
```

Thus, each pixel is mapped to a thread. In a thread, the cost aggregation is iteratively executed d times. The block size is dependent on the window size. A rectangular window is selected to use the already stored data in shared memory repeatedly along the epipolar line shown in Figure 3.

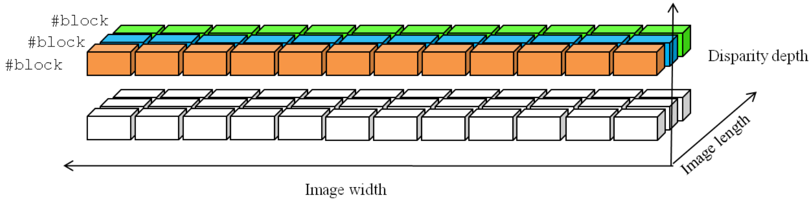

Fig. 3. Block tiling for the cost aggregation of the SAD method

In contrast, the semi-global cost aggregation is typically a serial computation from different paths. The cost optimization for each pixel in one direction requires the storage of both the computed cost values and the aggregated costs from the previously visited pixel. A pixel in the image contains $DisparityRange$ data elements in the cost cube, in which each concerned element of them maps to a thread. The block size is depended on the disparity range and the number of pixels inside each block is shown in the following kernel configuration:

```
dim3 ca_threads(PixelAmount, DisparityRange, 1);
dim3 ca_grid(ImageWidth/PixelAmount,1,1);
CostAggr<<<ca_grid, ca_threads>>>(CostCube, TempL, ...);
```

A further challenge is that pixels are no more independently with each other by the path wise aggregation. The optimized results backwards along a path are used for the actual optimization. The results must be rewritten into the global memory for the aggregation with other paths. Thus, the massive data accessing on global memory is not avoided completely. The ground idea for the parallelization is visualized in Figure 4. In this example, one image line is tiled in $ImageWidth/4$ segments. In a block there are $4 \times DisparityRange$ threads. The computation is then executed $ImageLength$ times iteratively in each block over the complete image.

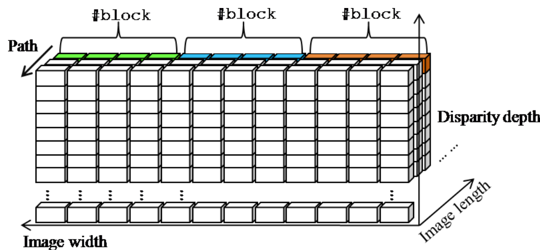

Fig. 4. Block tiling for the cost aggregation of the SGM method

Traditionally, the sweeping is executed more times for e.g. eight directions SGM. The fast implementation achieves the cost optimization in six directions

with two passes through the images. Cost aggregation can be extended for more directions, if the sweepings start from the other sides of the image. The incline optimizations e.g. path 1 requires the communication with other blocks. Hence, the optimized costs must be stored in the global memory. A distinct problem for large images is that CUDA features no block synchronization. This hardware inherency barrages the block communication on boundaries. The visibilities of the aggregations from oblique paths are depended on the amount of SMs. The quick approach and its problem is shown in Figure 5.

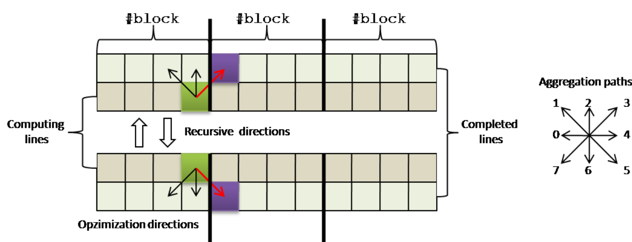

Fig. 5. Fast aggregation processing in three directions within one sweeping

The meanwhile optimized cost walls in the direction 0, 2, 4 and 6 are written to global memory, but not eliminated from shared memory, because they can be used for the next line. This finesse avoids reading the optimized cost from the global memory and economizes the expensive memory accessing for each block.

4 Results

The experimental 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 SMs on-chip and supports 1.3 CUDA compute capability [9]. The GPU implementations use the Middlebury Stereo Datasets [2] as well as aerial photos from the DLR's 3K system [1], which are additionally rectified on the GPU. Two comparisons are presented in this section: the run-time improvement of the GPU implementation with the CPU implementation and the accuracy and run-time differences between the SGM method and the SAD method.

Figure 6 shows the results of the Middlebury Teddy and Cones datasets using our SGM GPU implementation with an image size of 450×375 pixels. In addition, Figure 7 shows the results of the comparison between CPU and GPU implementation of an aerial image pair, which has an image size of 1000×1000 pixels with a disparity range of 80. Generally, they take similar results, but the disparities on the church roof on the GPU result are better than CPU execution, caused by the left to right and right to left cost aggregations on GPU use a different parameter with other aggregation paths. The compared CPU implementation runs on an Intel Core2 Q9450 CPU with 6 MB L2 Cache. The CPU

implementation needs about 5200 ms to finish the stereo processing including rectification. In contrast, the CUDA improvement requires 722ms for six aggregation directions and 1120 ms for eight aggregations 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. The GPU implementation is roughly 5 times faster than the CPU implementation written in C.

Fig. 6. Results of SGM GPU implementation using the Middlebury Stereo Datasets: the left images are related input images, the middle images are the ground truth depths and the right images are the GPU results

Table 1. Run-time comparison between CPU and GPU SGM implementation

	CPU run-time (8x)	GPU run-time (8x)
Rectification	432ms	96ms
Cost computation	200ms	9ms
Cost aggregation	4215ms	481ms(6×)/879ms(8×)
Disparity selection	362ms	136ms
Total	5209ms	722ms(6×)/1120(8×)ms

Figure 8 shows the run-times of the different steps of the SGM method for different image sizes. The run-time on small images with 384×288 pixels and a disparity range of 64 reaches 13 *fps*. Even though the results are promising, a potential improvement could be reached with a more adaptive data tiling strategy. The experiment with large images of 1000×1000 pixels exhibits a slowdown due to bandwidth latency. The path-wise cost aggregation is the key for the better performance of SGM, but results in an algorithm that cannot be parallelized as effectively as the local SAD method, as the temporarily aggregated

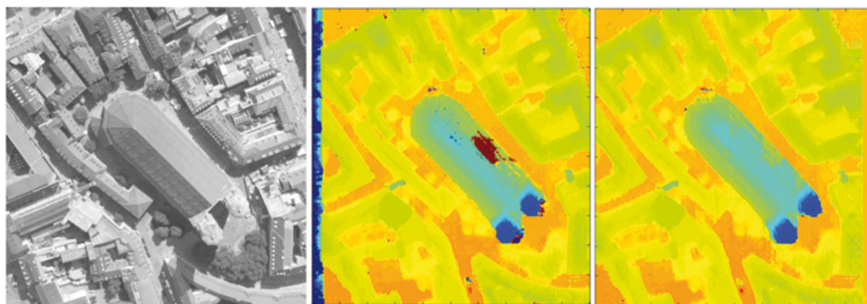

Fig. 7. Result comparison between CPU (middle) and GPU (right) SGM implementation. The left image is one of the related input images.

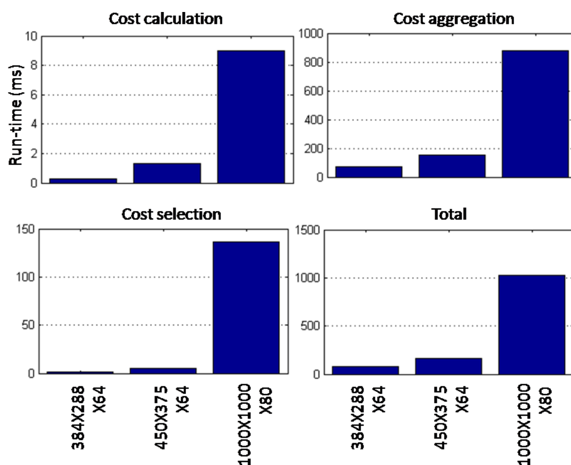

Fig. 8. The SGM GPU run-times on different image sizes

costs must be rewritten to the global memory in order to be used for aggregations with other directions.

The results of the SAD method with different window-sizes are compared with the SGM result and ground-truth in Figure 9. The SGM method achieves a less noisy result with more details. The results of SAD with a small window show many errors. The SAD with a larger window leads to a similar result as SGM, but it is still less precise. This comparison shows the improved reconstruction archived with the semi-global matching method.

With increasing window size, the local SAD method loses its runtime advantage with respect to SGM. Figure 10 shows strange spikes in the run-time graph with the window sizes in one dimension. The spikes at size 12 and 13 appear always using different test data and probably caused by hitting a cache limitation. After these impulses the factor of linear run-time increasing goes up and remains almost constantly.

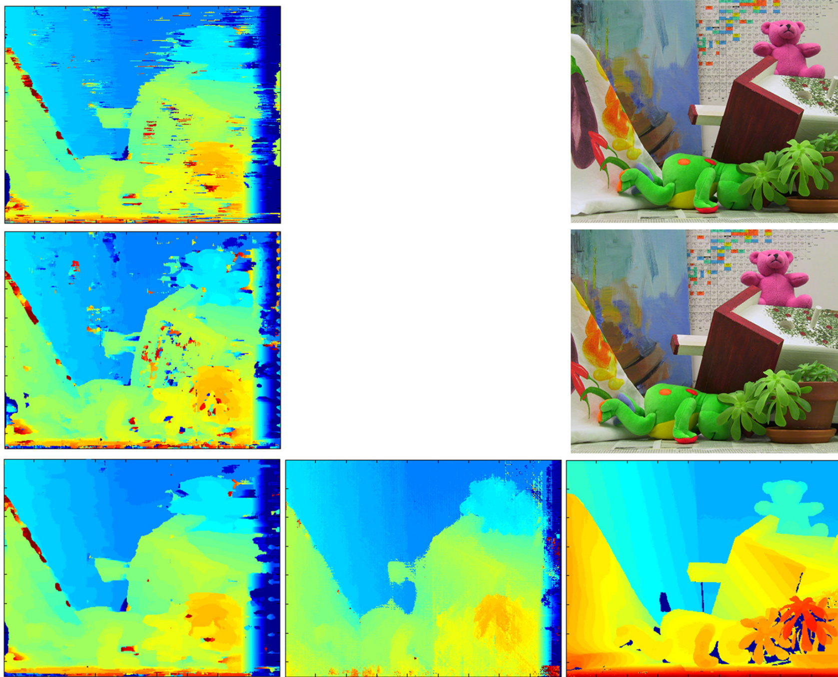

Fig. 9. The results compared between the SAD method with different window-sizes and the SGM result: the left part lists the SAD results with 3×31, 7×13 and 9×31 window size, respectively. In the middle is the SGM result shown. The input image pair and the disparity ground-truth are given on the right.

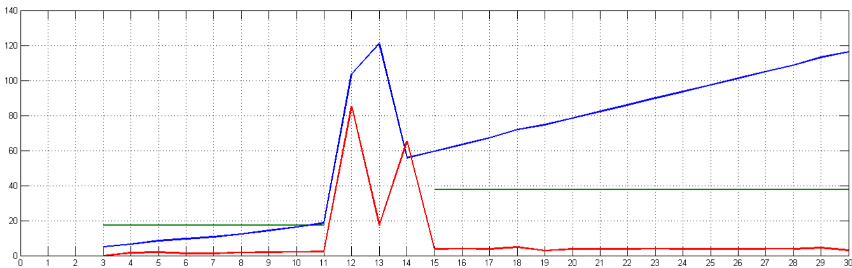

Fig. 10. The run-time analysis of a $3 \times x$ window of the SAD method: the blue line shows the processing time during the growing of window size in one direction. The red line demonstrates the run-time differences. The green line confirms the constantly run-time increasing before and after the impulse.

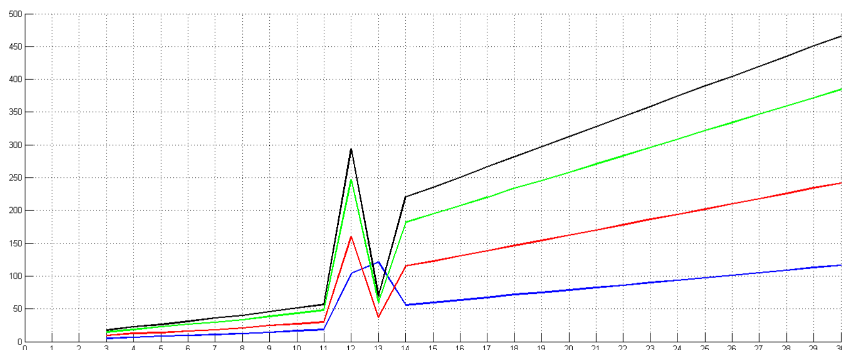

Fig. 11. The run-time analysis of the SAD method with different fixed window sizes in x direction: $3 \times x$ in blue , $5 \times x$ in red, $7 \times x$ in green and $9 \times x$ in black

In addition, the changes with different fixed window sizes in x direction are observed in Table 2 and shown in Figure 11. The comparison shows by the implementation of dense matching method on GPU, that the global methods keep their accuracy advantage and the cost/performance ratio of local matching methods is not beneficial for a fast processing on GPUs. The reason for inefficiencies of the local methods with large aggregation window sizes is that the computational win via the repeatedly employing of the updated data from the shared memory can not cover the memory accessing latency. Thus, the discret approached global methods like the SGM perform a better and more efficient result.

Table 2. Run-time increasing with fixed window sizes in x-direction and changed sizes in y-direction

	3x	5x	7x	9x	11x	13x	15x	17x	19x	21x	23x	25x	27x	29x
3x	4.9	8.4	10.0	14.3	18.6	121.0	59.5	67.1	74.7	82.2	89.9	97.3	104.9	113.2
5x	9.5	13.8	18.1	24.6	30.1	37.3	122.5	138.4	154.3	170.2	186.6	201.9	217.8	234.6
7x	14.6	22.7	29.3	38.3	48.1	59.5	194.6	220.0	245.2	270.6	295.9	322.0	346.6	371.9
9x	17.3	26.2	35.9	45.6	56.9	69.7	235.4	266.3	296.9	327.7	358.4	389.9	419.9	451.4

5 Conclusions

The proposed work demonstrates the general strategy for parallelization of dense matching methods on GPUs to show the potential capability of common graphics cards for general computation and to compare the implementations between local and global methods with the example of SAD and SGM method. The main architectural difference between CPU and GPU is the small amount of

fast shared memory and massive parallel computation power. This makes them suitable for simple problems without many dependencies between the data to be processed.

In contrast, the CPUs have large L2 Cache, which is enough to store the locally relevant cost cube. In future work, the dense matching methods will be optimized in CPU-implementation and compared with their GPU-optimized visions. The combination and adaptation between the current methods and modern hardware is not suitable. A parallel design of new method will be researched.

Thus, the SGM implementation on the GPU cannot use the full computation power of the graphics card. Future work will include the design of a (semi) global matching algorithm with a structure adapted to the constraints of the GPU hardware architecture.

References

1. Butenuth, M., Reinartz, P., Lenhart, D., Rosenbaum, D., Hinz, S.: Analysis of image sequences for the detection and monitoring of moving traffic. Photogrammetrie Fernerkundung Geoinformation 5, 421–430 (2009)
2. Middlebury Stereo Website (May 2010), `http://vision.middlebury.edu/stereo/`
3. Gehrig, S.K., Eberli, F., Meyer, T.: A Real-Time Low-Power Stereo Vision Engine Using Semi-Global Matching. In: Fritz, M., Schiele, B., Piater, J.H. (eds.) ICVS 2009. LNCS, vol. 5815, pp. 134–143. Springer, Heidelberg (2009)
4. Banks, J., Bennamoun, M., Corke, P.: Non-parametric techniques for fast and robust stereo matching (1997)
5. Krik, D.B., Hwa, W.W.: Programming Massively Parallel Processors: A Hands-on Approach (2010)
6. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International Journal of Computer Vision 47, 7–42 (2002)
7. Hirschmüller, H.: Stereo processing by semi-global matching and mutual information. IEEE Transactions on Pattern Analysis and Machine Intelligence 30, 328–341 (2008)
8. Hirschmüller, H., Scharstein, D.: Evaluation of stereo matching costs on image with radiometric differences. IEEE Transactions on Pattern Analysis and Machine Intelligence 31, 1582–1599 (2009)
9. NVIDIA. CUDA Programming Guide Version 3.0 (2010)
10. NVIDIA. OpenCL Best Practices Guide Version 1.0 (2009)
11. Fusiello, A., Trucco, E., Verri, A.: A compact algorithm for rectification of stereo pairs. Machine Vision and Applications 12, 16–22 (2002)