Buletinul Ştiinţific al Universităţii "Politehnica" din Timişoara

Seria ELECTRONICĂ şi TELECOMUNICAŢII
TRANSACTIONS on ELECTRONICS and COMMUNICATIONS

Tom 55(69), Fascicola 2, 2010

Disparity Map Computation Speed Comparison for CPU, GPU and FPGA Implementations

Adrian Leu[1], Dan Bacără[2], Ioan Jiveţ[3]

Abstract – In this paper a comparison of the processing speed of the disparity map computation using a CPU, a GPU and an FPGA is presented. First the straight-forward implementations of the block matching algorithm for the CPU and GPU are presented, followed by the newly developed architecture for FPGA implementation. The GPU used in this paper is an Nvidia Tesla C1060, programmed using the Nvidia CUDA API. The sum of absolute differences (SAD) has been chosen to compute the matching cost for the block matching algorithm, because of its simplicity, which facilitates a hardware implementation and makes the algorithm suitable for use in applications where a high frame rate is required. The last part of the paper presents a comparison between the processing speeds of the three considered devices.

***Keywords* — high speed disparity computation, SAD block matching algorithm, Tesla GPU, CUDA, FPGA implementation**

I. INTRODUCTION

Stereo vision has become a very important field of image processing because of the great benefit it offers by facilitating the computation of the 3D location of objects from the scene using a pair of images. The depth information can be useful for object grasping in applications like service robotics [1][2], collision avoidance for autonomous robots [3] or object detection in driving assistance systems [4][5][6]. The depth information is computed using correspondence points from the two images.

The focus in this paper is on the block matching method, using the sum of absolute differences (SAD) as a matching cost function. Since the implementation for a CPU is not fast enough for applications where a high frame rate is required, the acceleration using a GPU was investigated, as well as the possibility of implementing the algorithm using an FPGA. The used GPU was an Nvidia Tesla C1060, programmed using the Nvidia CUDA API [11].

A various set of images has been tested, with the main focus on images of street scenes having a resolution of 1280x480 pixels and a pixel depth of 8 bit. The stereo camera used to capture these images has a baseline of 456 mm, a focal length of 11mm and square pixels with a width of 12 µm.

The paper is organized as follows: section II describes the disparity map computation, in section III the CPU, GPU and FPGA implementations are described, section IV shows a comparison in the execution time for the three devices and section V presents the conclusions and future work.

II. DISPARITY MAP COMPUTATION

For a general stereo camera setup the correspondent point in the right image for an image point in the left image lies along the correspondent epipolar line [7]. If the optical axes of the two cameras are parallel, the epipolar lines will be parallel to the horizon, therefore simplifying the process of finding the correspondence point. In this case, the searching process takes place along one image line and consists in finding a pixel having the closest intensity value to the intensity of the reference pixel. Even though this simplifies the process, uncertainties can appear if more pixels on the same line in the right image have the same intensity value.

In order to overcome this problem, area based methods like block matching [8] have been developed. The idea behind them is to use the pixel neighbourhood for finding the correspondence pixel, therefore minimizing the probability of a wrong match through the fact that a certain number of neighbouring pixels is less likely to match more regions in the correspondence image. The only problem appears for uniformly textured objects, for which the correspondence is still ambiguous.

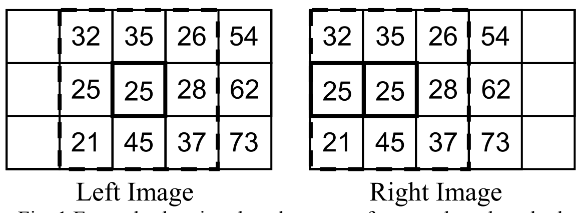

Fig. 1 Example showing the advantage of an area based method compared to pixel matching

[1] Institute of Automation, University of Bremen, e-mail: aleu@iat.uni-bremen.de
[2] Facultatea de Electronică şi Telecomunicaţii, Departamentul Electronică Aplicată, e-mail: dan.bacara@gmail.com
[3] Facultatea de Electronică şi Telecomunicaţii, Departamentul Comunicaţii, e-mail: joan.jivet@etc.upt.ro

Fig. 1 shows an example of an image pair for which the disparity map should be computed. The current pixel is the central pixel in the left image, which should be matched with a pixel in the right image. The images are considered to be rectified, so the correspondence pixel is on the same line in the right image. There are two pixels in the right image having the same intensity value. This ambiguity can be solved if neighbouring pixels are also used. If the 3x3 window around the considered pixel is used, the corresponding region can be uniquely identified in the right image.

The presented example raises the question of how big the window size of neighbouring pixels should be. If the window is small, like 3x3 or 5x5, it is possible that the ambiguity problem cannot be reliably solved and that the resulting disparity map contains much noise. If the window is big, like 19x19 or 21x21, there will be less noise, but the time needed to process the whole image would be much bigger and small objects might be completely dropped. This means that the choice of window size is application and image resolution dependent.

In order to illustrate the difference in resulting disparity maps for different window sizes, Fig. 2 shows the disparity map computed for a 5x5 window and Fig. 3 shows the result for a 19x19 window. In both cases the original image pair has a resolution of 1280x480 pixels and the maximum disparity is 64.

It can be noticed that both images have a black frame that is wider in the left side. This is the result of using windows for matching and therefore the first and last pixels in each row and column do not have enough neighbours to form a complete window and are dropped. The big number of missing pixels on the left side is caused by the used maximum disparity, since all reference pixels must have the same number of correspondence pixels to be checked.

Certain reliability tests can be performed in order to analyze if the match is reliable. One way is to use a threshold to exclude pixels for which the match was not good enough. Another way is to perform a left to right and right to left match to detect occluded regions that might cause noise on the resulting image. This operation consists in computing the disparity map first with respect to the left image then with respect to the right image and eliminating pixels from the resulting image for which the left to right and right to left disparity is not the same.

In order to accelerate the computation, a priori information can be used. If the optical axes of the cameras are parallel, an object will always appear in the right image shifted to the left, compared to its location in the left image.

In Fig. 4 it can be seen that the two objects in the right image appear shifted to the left compared to the left image. This information can be used to reduce the searching area to pixels located to the left of the location of the reference pixel.

It can also be seen that the cylinder shifted more than the cube, which means that it is closer to the camera. The disparity is a measure that shows how much an object appears shifted in the right image compared to the left image. The following formula illustrates this, in which *d* is the disparity and X_L and X_R represent the coordinates of the pixel on the x axis in the left and right image.

$$d = X_R - X_L \qquad (1)$$

The distance from the camera to an object's plane is inversely proportional with the object's disparity. The relationship between the distance and the disparity is:

$$D = \frac{B \cdot f_p}{d} \qquad (2)$$

In this formula, *D* represents the distance from the camera to the object plane, *B* is the stereo camera base line, f_p is the focal length of the camera and *d* is the object's disparity. The focal length has to be converted to pixels using the following formula:

$$f_p = \frac{W_i \cdot f}{W_s} \qquad (3)$$

In this case, f_p is the resulting focal length in pixels, f is the focal length in mm, W_i is the image width in pixels and W_s is the sensor width in mm.

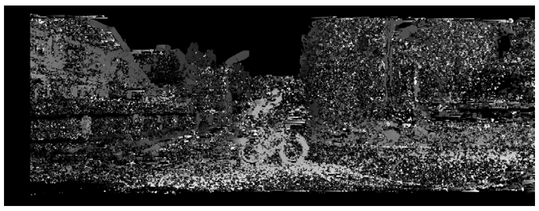
Fig. 2 Disparity map computed for a 5x5 window

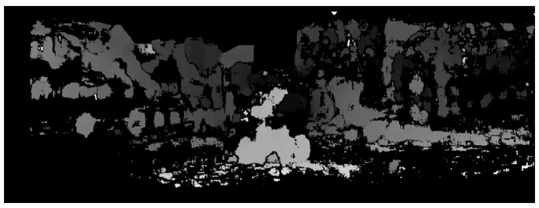
Fig. 3 Disparity map computed for a 19x19 window

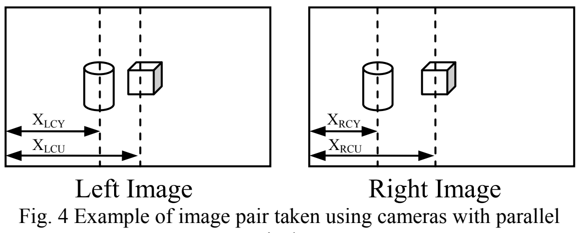

Fig. 4 Example of image pair taken using cameras with parallel optical axes

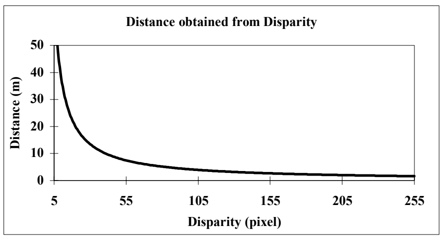

Fig. 5 Distance calculation from disparity values

The plot shown in Fig. 5 has been computed using the values for base line and focal length that have been presented in the introduction. For small disparity values, if for example the disparity value changes with one pixel, the change in distance is very significant. The conclusion at this point is that the used camera system gives reliable distance information, within the acceptable tolerance of 5%, only for distances up to 20m. Since objects closer than 6.5m are not interesting for this application, the maximum computed disparity is 64.

In conclusion for the particular application presented in this paper, the window size for the block matching should be around 19x19 and the maximum disparity 64. Also the minimum required disparity is 20, so the total number of disparity levels to be computed is 44.

III. IMPLEMENTATION

In this section the implementation of the block matching algorithm using SAD as the matching cost will be presented. First the pseudocode of the algorithm will be shown, followed by the straight-forward CPU implementation, and then the GPU and FPGA implementations will be briefly described.

The pseudocode for the block matching algorithm is simple and easy to understand. Basically for each pixel in the left image, the corresponding pixel in the right image must be found, using the sum of absolute differences as a matching cost.

The idea is to iterate through each pixel in the image, compute the sum of absolute values (SAD) for the entire window having the considered pixel in the centre. The window in the right image is then shifted one pixel to the left and the SAD value is computed again. This operation is repeated until all disparity levels have been analyzed. The resulting disparity value is obtained for the disparity level that generated the minimum SAD value.

The pseudocode for this algorithm can be seen below.

```
FOR Y = MIN_Y to MAX_Y
    FOR X = MIN_X to MAX_X
        IDX = Y * IMG_WIDTH + X
        MIN_SAD = MAXINT
        DISP = 0
        FOR D = MIN_D to MAX_D
            XL = X
            XR = X - D
            CURR_SAD = computeSAD (XL, XR, Y)
            IF CURR_SAD < MIN_SAD
                MIN_SAD = CURR_SAD
                DISP = D
            END IF
        END FOR
        DISP_IMG (IDX) = D
    END FOR
END FOR
```

The X and Y limits are obtained by taking into consideration only the pixels that have all required neighbours, considering the maximum disparity and the size of the SAD window:

```
MIN_X = MAX_DISP + SAD_SIZE/2 – 1
MAX_X = IMG_WIDTH - SAD_SIZE/2 – 1     (4)
MIN_Y = SAD_SIZE/2
MAX_Y = IMG_HEIGHT - SAD_SIZE/2 – 1
```

The minimum and maximum disparities are obtained from the application requirements. In the considered application `MIN_D = 20` and `MAX_D = 64`.

The *computeSAD* function computes the sum of absolute differences for a window in the left and a window in the right image. It takes as an input the location of the pixels to be matched, which are in the centre of the window to be matched. These pixels are on the same y coordinate in the case of rectified images taken with cameras having parallel optical axes and therefore only one y coordinate needs to be passed. `XR` is the location of the considered pixel in the right image and is computed by subtracting the currently analyzed disparity from the x coordinate of `XL`, which is the x coordinate of the reference pixel from the left image.

CPU

The straight-forward CPU implementation using a single thread is easy to deduce from the pseudocode and will not be described in detail. If more than one thread is used, the image must be divided, so that each thread operates on a different region of the image, therefore finishing the task faster. This division does not necessarily mean copying parts of the image into different memory locations, since this would create additional overhead. The same result can be obtained by using more pointers on the same memory location, one for each thread. Although more threads run in parallel would finish the operations much faster, the memory access can easily become a bottleneck of the application if all threads access the same memory location. However, the straight-forward CPU implementation is easy to do and is a good way to get acquainted with the disparity map computation.

GPU

In this section a straight-forward GPU implementation using the Nvidia CUDA API will be briefly described. The resulting code has been run on an Nvidia Tesla C1060 device.

The big difference between CPU and GPU programming is the number of cores. A regular CPU has 2-4 cores and a good CPU has up to 16 cores, while the Nvidia Tesla C1060 GPU has 30 multiprocessors (MP) of 8 cores each, summing up to 240 scalar processor cores (SP) [12].

The software architecture consists of threads, which are grouped into blocks. One block can contain a maximum of 512 threads and will be executed on one MP. A warp, consisting of 32 threads, will be physically executed in 4 clock cycles in the MP and a scheduler switches between warps. The warp is the smallest possible execution unit in CUDA. The maximum total number of threads is 65536. The only constraint is that at the same time all threads will execute the same operation according to the Single Instruction, Multiple Data (SIMD) principle [10]. Also, all threads in one block have access to the same shared MP memory, which has a higher bandwidth compared to the bandwidth of the global memory and can be used to accelerate the computation.

In conclusion, the algorithm must be parallelized in order to efficiently run on a GPU. The simplest way of doing this is to let each thread process only a few pixels. The current application has images of 1280x480 pixels with a pixel depth of 8 bit. The considered maximum disparity is 64 and the window size 19, which means that the effective size of the image that has to be processed, according to the relations (4) is 1199x461. A warp consists of 32 threads, so the total number of threads in a block must be a multiple of 32 and not exceed 512. The maximum total number of threads is 65536, so there can be a maximum of 128 blocks if each block has 512 threads. The straight-forward implementation uses 128 blocks of 480 threads each. Since 461<480, an entire image column can be processed by a block at a time. Since there are 128 blocks and the image is 1199 pixels wide, each block should process a total of 10 image columns.

```
IDXB = BLOCK_IDX
IDXT = THREAD_IDX
TB = TOTAL_BLOCKS
Y = IDXT
FOR C = 1 to 10
    X = IDXB * TB + C
    IDX = Y * IMG_WIDTH + X
    MIN_SAD = MAXINT
    DISP = 0
    FOR D = MIN_D to MAX_D
        XL = X
        XR = X - D
        CURR_SAD = computeSAD (XL, XR, Y)
        IF CURR_SAD < MIN_SAD
           MIN_SAD = CURR_SAD
           DISP = D
        END IF
   END FOR
   DISP_IMG (IDX) = D
END FOR
```

In the presented pseudocode, the changes with respect to the CPU version are presented in bold. `BLOCK_IDX`, `THREAD_IDX` and `TOTAL_BLOCKS` are values delivered by the CUDA API. It can be seen that the for loops have been replaced by block and thread indexes, which indicate which thread of which block is currently accessing the function. The only remaining loop is the column loop which goes from 1 to 10 for the considered images. A remark has to be made here. Considering the fact that there are 1280x480 operations, from which only 1199x461 are valid, in the *computeSAD* function it must be checked if the current pixel is inside the valid domain.

It can be seen that the pseudocode for the GPU implementation is not very different from the pseudocode for the CPU implementation, but the way of thinking as well as the underlying hardware are completely different.

FPGA

This section presents a straight-forward FPGA implementation, which differs considerably from the implementations presented so far. This is because the FPGA is highly parallel and is not a processing unit, but rather a collection of configurable components. Through this high parallelism, high processing speeds can be achieved for specialized operations. In this paper, a Xilinx Spartan 3A DSP 1800 FPGA [13] has been used for implementation, because it is a relatively low cost development board that still offers a lot of resources and also includes a VGA output for displaying the results on a monitor.

The FPGA implementation has been done for images of 128x128 pixels with a pixel depth of 8 bit, a maximum disparity of 16 and a SAD window size of 7x7.

Fig. 6 shows an overall view of the FPGA implementation of a SAD block. The left and right images are stored in dual port ROM memories for the developed offline application in order to display them on the screen at the same time as the disparity map is computed. These memories however can be easily replaced by buffers in which the pixels coming from the camera can be stored for the online application,

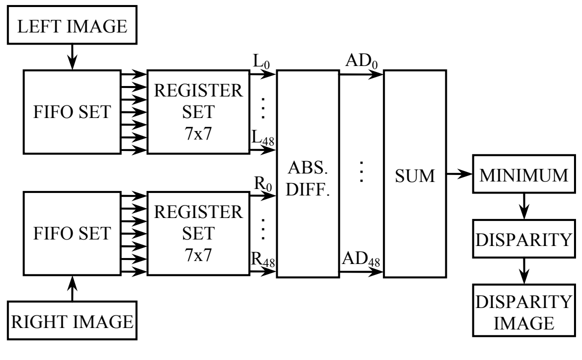

Fig. 6 Global view of the FPGA implementation of one SAD block

because at the end only the disparity map is needed and the original images don't have to be displayed using the FPGA's VGA port.

While in the case of a CPU as well as for the GPU the reading from memory is self-evident and does not have to be explained, for the FPGA implementation this plays an important role and must be explained. The image rows are read from the left and right images into First-In-First-Out (FIFO) buffers. This is done in order to speed up the access to the pixels from the next SAD window, because the external memory access is very slow compared to the memory located in the FPGA chip.

The output of the FIFO set is shifted into registers in order to allow access to all 49 values at the same time. After the FIFO set is full, 7 clocks later the registers will contain the data needed to compute the first SAD window. From that moment, for each clock cycle the right FIFO will shift to the right, moving a new window column into the register, instead of re-reading the whole window from memory. This operation continues until the maximum disparity has been reached. At this point the left FIFO shifts one window column into the register, while the right FIFO is reloaded.

The 49 absolute differences are computed in parallel and then summed in a pipeline manner, so that after the initial latency, the result of a SAD window is obtained every clock cycle.

The minimum block checks if the currently computed SAD value is the minimal value so far for the given reference window in the left image. If yes, the new value is saved together with the offset that represents the actual disparity. At the end of a complete cycle, the index reaches the maximum disparity and the disparity block will contain the disparity value for the current reference pixel. This value will be copied to the disparity image, which is stored in a dual port RAM. The dual port has the same purpose as in the case of the ROM: allowing the image to be displayed on the monitor while it is computed.

The presented structure occupies 1823 slices on the Spartan 3A board. Since the board has 16000 slices, it is possible to use 8 such structures in parallel in order to speed up the process. The final architecture can be seen in Fig. 7.

There are 8 SAD blocks that can process 8 image lines in parallel, which means that the overall processing will be 8 times faster. However the performance of the algorithm can be improved if it is implemented on a board with more resources, since the number of SAD blocks only depends on the FPGA resources.

If many blocks are used, the memory can become a bottleneck. This can be solved by simultaneously writing the same information in different memory blocks and only a limited number of SAD blocks read the information from one memory block.

The used Xilinx FPGA has a 25.175 MHz clock that must be used if the VGA output is used to display the images and a 125 MHz clock that can be used if the images don't have to be displayed using the on-board VGA output.

IV. PROCESSING SPEED COMPARISON

In this section the comparison in processing speed between the CPU and GPU and between the CPU and the FPGA will be presented. The comparison between CPU and GPU has been done using 1280x480 images with a pixel depth of 8 bit and the images used to compare the CPU and the FPGA have a resolution of 128x128 and an 8 bit pixel depth. Since the processing time is long, only one frame has been processed for the CPU and 10 frames for the GPU. The mean processing time was then computed for the 10 frames. The values for the FPGA were obtained by counting the clock cycles required to complete the operation and dividing the value by the frequency of the used clock. The number of clocks required was obtained using Model SIM. The algorithm implemented for the CPU runs on a single core on a CPU at 2.6 GHz, the GPU algorithm runs on the 240 cores of the Tesla C1060 at 1.3 GHz and the FPGA implementation runs either using the 25.175 MHz VGA clock or the 125 MHz system clock.

Fig. 8 shows the processing time required by the CPU to compute a disparity map for a pair of input images, while Fig. 9 shows the processing time required by the GPU to produce the same result. The legend shows the computed disparity levels followed by the considered SAD window size.

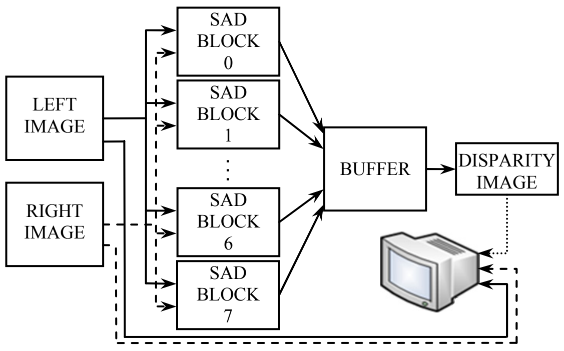

Fig. 7 Global view of the FPGA implementation

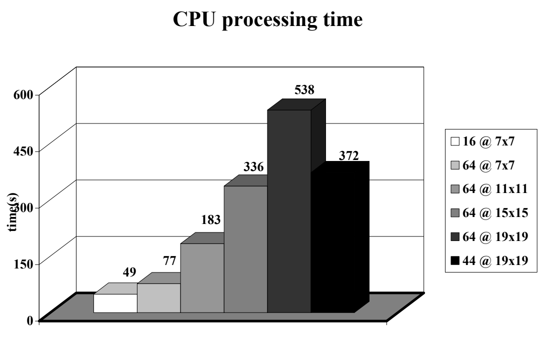

Fig. 8 Processing time using the CPU

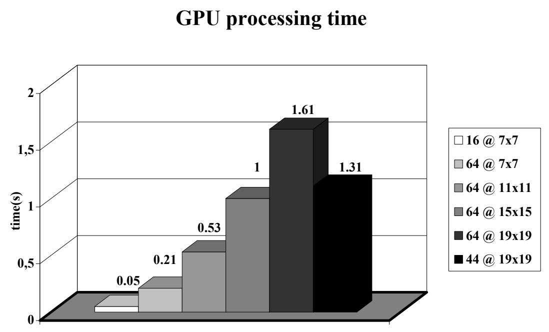

Fig. 9 Processing time using the GPU

It can be seen that the processing time increases with the chosen SAD window size. If only 44 depth levels are processed, as explained at the end of section II, the required time to compute the disparity map is shorter with 30% for the CPU and 20% for GPU. Overall, the computation on the GPU finishes 300-400 times faster than on the CPU, which makes the GPU implementation useful if speed is important, but the high cost of the Nvidia Tesla C1060 might not be optimal for a low-cost system, especially if a PC is not already part of the system.

Fig. 10 shows the comparison between the achieved frame rate using the CPU and the FPGA implementation. The maximum considered disparity was 16 and the SAD window was of 7x7. It can be seen that the FPGA completes the computation much faster than the CPU, therefore being a good solution for a low-cost compact system, without even requiring a PC.

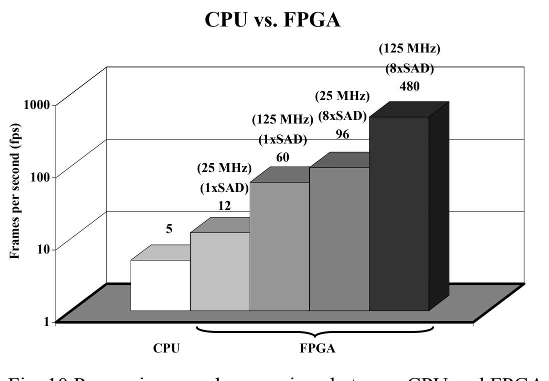

Fig. 10 Processing speed comparison between CPU and FPGA

V. CONCLUSIONS AND FUTURE WORK

In this paper a computation speed comparison between straight-forward implementations of the block matching disparity map computation using the sum of absolute differences (SAD) as the matching cost for CPU, GPU and FPGA is presented. For the CPU and GPU implementations the pseudocode is given that explains the algorithm and the parallel processing ability of the GPU. The FPGA implementation is described in detail, including the possibility of improving the computation speed by using an FPGA with more resources.

For all three implementations all the differences and sums are computed for each block that has to be matched, even for the FPGA, where the memory access has been optimized by using FIFO buffers. A way of speeding up the computation is to reduce the number of differences and sums computed for every block by reusing some of the values that have been computed for previous blocks. Besides this, the memory tends to become a bottleneck in all three cases. In the case of the CPU, processor registers can be used to store intermediate results for speeding up the process, while in the case of the GPU shared memory, as well as registers can be used for this purpose. The improvements for the FPGA might include a better usage of the data from the FIFO sets and also a reuse of values that have already been read from memory.

VI. REFERENCES

[1] Grigorescu S.M., Prenzel O., Gräser A. (2010). Model Driven Developed Machine Vision System for Service Robotics. 12th International Conference on Optimization of Electrical and Electronic Equipment, OPTIM 2010. pp. 877 - 883
[2] Li L., Kohl Y.T., Gel S.S., Huang W. (2004). Stereo-Based Human Detection For Mobile Service Robots. 8th International Conference on Control, Automation, Robotics and Vision Kunming, China. Vol 1. pp. 74 - 79
[3] Saurav K. (2009). Binocular Stereo Vision Based Obstacle Avoidance Algorithm for Autonomous Mobile Robots. IEEE International Advance Computing Conference (IACC 2009). pp. 254 - 259
[4] Talukder A. and Matthies L. (2004). Real-time Detection of Moving Vehicles using Dense Stereo Objects from Moving and Optical Flow. Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 3718-3725
[5] Nedevschi S., Vatavu A., Oniga F., Meinecke M. M. (2008). Forward Collision Detection using a Stereo Vision System. International Conference on Intelligent Computer Communication and Processing, 2008. ICCP 2008. pp. 115 - 122
[6] Nedevschi S., Danescu R., Marita T., Oniga F., Pocol C., Sobol S., Tomiuc C., Vancea C., Meinecke M. M., Graf T., To T. T., Obojski M. A. (2007). A Sensor for Urban Driving Assistance Systems Based on Dense Stereovision. Proceedings of the 2007 IEEE Intelligent Vehicles Symposium. pp. 276-283
[7] Papadimitriou D. V. and Dennis T. J. (1996). Epipolar Line Estimation and Rectification for Stereo Image Pairs. IEEE Transactions On Image Processing 5(4):672-676
[8] Tao T., Koo J. C., Choi H. R. (2008). A Fast Block Matching Algorthim for Stereo Correspondence. IEEE Conference on Cybernetics and Intelligent Systems. pp. 38-41
[9] Hadjitheophanous S., Ttofis C., Georghiades A. S., and Theocharides T. (2010). Towards Hardware Stereoscopic 3D Reconstruction A Real-Time FPGA Computation of the Disparity Map. Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1743 - 1748
[10] Cockshott, W. P., Renfrew, K. (2004). SIMD programming manual for Linux and Windows, Springer
[11] http://www.nvidia.com/object/cuda_home_new.html
[12] http://www.nvidia.com/object/product_tesla_c1060_us.html
[13] http://www.xilinx.com/products/devkits/HW-SD1800A-DSP-SB-UNI-G.htm