Accelerated Stereo Vision Using Nvidia Jetson and Intel AVX

Imran A. Syed[1](✉), Mandar Datar[2], and Sachin Patkar[2]

[1] Center for Artificial Intelligence and Robotics, DRDO, Bangalore, India
imransyed@cair.drdo.in
[2] Department of Electrical Engineering, IIT Bombay, Mumbai, India

Abstract. Stereo vision is a low cost and passive mechanism to perceive the environment for robotic applications. The huge compute requirements of stereo vision algorithms have been a major challenge for their usage in real world applications on small robots. Standard stereo depth estimation algorithms Sum of Absolute Differences (SAD), census transform and an advanced algorithm Semi-Global Matching (SGM) are discussed in this work. This paper presents novel real time implementation of these three stereo vision algorithms on two different compute platforms *i)* Intel AVX (Advanced Vector Extension) and *ii)* Nvidia Jetson GPU (Graphical Processing Unit). The Intel CPU implementation of stereo algorithms is optimized by using OpenMP (Open Multi-Processing) for multi-threading, AVX registers for vectorization and several other novel ideas for real time processing. Nvidia Jetson implementation is efficiently designed for maximum speed-up on a low end GPU such as Jetson TK1. Post processing steps such as local extrema detection, left-right consistency and median filter are used to improve the final disparity image. We have achieved speedup of the order of 30x when compared with naïve CPU implementation.

Keywords: Stereo vision · Depth estimation · Intel AVX · Nvidia Jetson · GPU acceleration

1 Introduction

1.1 Stereo Depth Estimation

Stereo cameras are low cost and low SWAP (Size, Weight, And Power) devices due to which they are used extensively in Autonomous aerial and ground robotic vehicles for environment perception. Stereo cameras give a 3D perception of the environment which can be used for applications such as Simultaneous Localisation and Mapping (SLAM) [1], 3D obstacle avoidance [2], object detection etc. Depth estimation from stereo cameras is one of the preliminary steps required for any of these 3D perception algorithms. This is a computationally expensive problem and processing requirements increases linearly with image size. To reduce the computation time, multi core CPUs or General Purpose Graphical Processing Units (GPGPU) are used but have high power requirements. Due to power constraints on the robotic platforms, low power embedded systems such as Jetson GPU and FPGA boards have become popular due to their massive parallelization capability.

© Springer Nature Singapore Pte Ltd. 2021

In this work, three algorithms have been attempted, Sum of Absolute Differences (SAD), CENSUS & Semi-Global Matching (SGM) algorithms, which take in calibrated and rectified [4] stereo pair as input and produce a output disparity image which can be converted to depth image using calibration parameters. SAD & CENSUS are simple stereo algorithms that have lower accuracy but much faster in terms of compute whereas SGM is complex algorithm that has higher accuracy but slower in terms of compute. More complex deep learning based algorithms exist but low power real time implementation of such algorithms is a challenge. SGM algorithm is being adapted and published even in the recent journals owing to its robustness in varied environments.

1.2 Hardware Selection

For real time implementation of depth estimation algorithms from stereo cameras, various hardware options are considered such as using (i) GPU cards in a desktop computer, (ii) FPGA boards, (iii) custom embedded boards, and (iv) exploiting the multi core architecture of commercially available multi-core CPUs. GPU cards support massive parallelization but have huge power requirement which inhibits their usage on small platforms such as drones etc. Implementations of stereo algorithms on high end GPUs exist in literature but they have not been adapted or fine-tuned for low powered devices and hence do not perform as efficiently as our implementation. The second alternative FPGA boards have been explored for stereo cameras and have shown promising results as detailed in the literature survey. Custom products for aerial platforms are available with FPGA hardware for stereo depth estimation. The third alternative is to use specialized embedded boards such as Nvidia Jetson which are low power devices with GPU capabilities. These boards are used extensively for small robotic platforms. We have chosen one such board TK1 from Jetson family to prove the real time implementation on mobile platforms with low power. The fourth alternative is to use existing CPUs for depth estimation. Commercially available multi core CPU architecture supports vector processing of instructions and multi-threading capabilities, which can be exploited for fast processing without the use of any extra hardware. In this paper we have established stereo depth estimation on Intel Xeon CPU which can be extended to any Intel CPU with Advanced Vector Xtension (AVX) [3] registers. In this work we have implemented on Intel Xeon CPU and Nvidia Jetson independently. Nvidia Jetson is based on NVIDIA's mobile processors Tegra, which is a System on Chip (SoC) with ARM+GPU+ISP custom made by Nvidia for integrated processing in low power devices. Jetson TK1 has 2 GB integrated memory with quad core ARM processor A15 as well as 16 GB onboard memory. Compute Unified Device Architecture (CUDA) programming language, developed by Nvidia, is used for programming the Jetson device.

This paper presents a novel implementation of the stereo SGM algorithm proposed in [10] with several ideas to reduce the computation time but not modifying the algorithm itself. Some of these ideas have been explored already by other papers such as OpenMP multi-threading, two pass approach, and vectorization. However, these ideas have not been able to reach real time speeds. Additional ideas such as recursive XOR for minima computation, AVX register usage, usage of shared GPU memory, CUDA shuffle instructions and better memory coalescing has led to very fast depth estimation. Novelty of this paper lies in the implementation rather than the stereo algorithms. The ideas presented in this work are not specific to SGM algorithm and can be extended to other such algorithms as well but require detailed study and analysis of the algorithm. Minima computation, two-pass approach, and vectorization are generic in nature, which can be applied to any parallelization framework. But memory coalescing and CUDA shuffle instructions have to be carefully designed for specific algorithms.

2 Literature Survey

Stereo depth estimation papers date back from 1960 with accurate and faster solutions being published even now. Stereo matching essentially finds the shift of each pixel in left and right images on the corresponding horizontal epipolar lines. There are three types of approaches for stereo depth estimation; local, global and machine learning (ML) based.

In local approaches, a small window is selected around the pixel to match with the candidate pixels in right image. The matching cost between two windows can be computed using various distance measures such as Sum of Squared Distances (SSD), Sum of Absolute Distances (SAD), Normalized Cross Correlation (NCC) [5]. Census Transform [6] distance measure encompasses the contrast information and is computationally less expensive. Instead of a fixed window size, adaptive windows [7], shifted window [8], hierarchical windows [9] have been proposed to improve the accuracy. Global methods use constraints on the whole image and optimize a global cost function to generate a disparity map and hence tend to be more accurate than the local methods. SGM [10] is a global stereo disparity estimation method based on global mutual information cost function minimization. Various versions of this method are still among the top performing stereo algorithm on Middlebury datasets. In Graph cut method [11] and Belief propagation methods are other such popular global methods. A common problem with global methods is the large computation time as they involve solving a global optimization problem. The paper [18] implements NCC based local window matching algorithm (similar to SAD/CENSUS as discussed in this paper) on Jetson TX2 (newer version of Jetson TK1).

Recent trend in stereo depth estimation problem is shifting towards Machine learning. The main difficulty with machine learning approach is that it requires large labeled datasets with ground truth. To solve this problem a large synthetic dataset [12] is developed for training the neural network. FlowNet [13] was trained on this simulated data but surprisingly performed well on real world datasets. In another approach [14], self-learning approach is used where image warping loss function is used instead of a labeled ground truth. However, these ML approaches requires GPU with 4 GB memory per image and more than 100 W GPU power usage and optimizing these approaches is a challenge in itself as deep learning model have become increasingly complex over time and the usage of multiple libraries which are generic in nature and are less optimized for that reason.

Optimization of the stereo depth estimation methods on the latest compute hardware is an alternative solution to this problem. NCC based block matching is implemented in real-time [15] with good results but NCC is a simple block matching technique with low accuracy when compared with SGM. SGM algorithm is accelerated using FPGA hardware [16] to achieve 5 fps. M/s DJI, which is an aerial drone manufacturing company, has produced a product Guidance Kit, which uses FPGA for stereo depth estimation. Even though FPGA is low powered and show promising results, FPGA development is costly and not easily available for low production. General purpose GPUs have massive parallelization capability and are used for accelerating stereo vision in multiple applications but their power requirements have been a hindrance for their application on small robotic platforms. Embedded implementation of SGM on Jetson has been explored in [17] with focus on GPU implementation but implementation on CPU has not been discussed. Optimization of the algorithm using vector registers and multi-threading for use on Intel architectures and low powered GPUs has not been explored in literature as per our knowledge.

3 Our Implementation

For real time implementation of stereo depth estimation, we have shown on two different platforms. Nvidia Jetson was chosen due to its GPU, low power requirements and portability on mobile robots. Intel CPUs were chosen due to their ubiquitous presence and modern multi-core architectures. We implemented stereo algorithm in real time on Intel Xeon CPU without use of any extra hardware.

3.1 Intel CPU Optimization

The main challenges were the shifting window approach and 8-directional cost update step. The shifting window approach involves repeated memory access and computation on the same pixel. Cost update step involves aggregating cost from 8 different directions as per the update equation which requires computing the minimum and aggregating for each pixel across disparity space. These challenges are addressed using OpenMP and AVX registers of the CPU as well as algorithmic changes such as dividing the cost computation in two steps for saving memory access, recursive XOR for computing minimum in logarithmic complexity, and vectorization of the instructions, which are detailed in this section. This has led to a speedup of $\sim$50x on a Intel Xeon processor.

SAD Algorithm. For each pixel in the left image, a fixed number of candidates (N) are chosen in the right image. This fixed number is called disparity range. For each candidate window, the SAD distance measure is computed with the reference window. In case of serial implementation, the absolute differences and summations are recomputed multiple times for neighboring pixel since neighboring blocks have almost the same reference blocks. Moreover, this leads to repeated memory access of same element. In our approach, all the absolute differences are computed in first pass, by shifting the right image successively by one pixel and saving the stack of difference images as shown in Fig. 1. The computation of absolute difference for each shift is done using AVX instructions which compute 8 pixels in one instruction. Also multi-threading is used to parallelize the processing so that each thread works on fixed number of columns depending on the available cores. In the second pass, for each image of the stack, a box filter is applied to compute the summation. Then the minimum is chosen across stack for each pixel. This improves the speed from naïve approach by $\sim$15x.

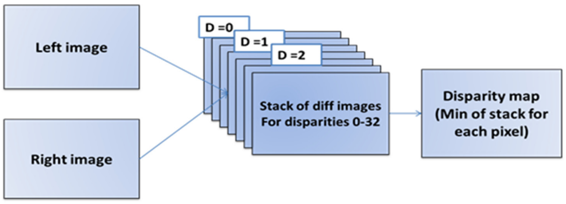
Left image
Right image
D =0
D =1
D =2
Stack of diff images For disparities 0-32
Disparity map (Min of stack for each pixel)

Fig. 1. SAD optimization

CENSUS Algorithm. For each pixel, a CENSUS vector is computed for every window and hamming distance measure is computed with reference window. Since CENSUS uses contrast information in the window, it eliminates sensitivity to absolute intensity and to outliers. In naïve implementation, neighbouring pixels compute CENSUS vectors repeatedly for same reference pixels. This is optimized by dividing the process in two passes. In the first pass, all CENSUS vectors are computed for both images, using multi-threading and vectorized instruction set and saved as 64-bit integers(window size of 9×7 is chosen). In the second pass, hamming distance is computed between candidate pixel and reference pixels. 128-bit AVX registers are used for computing hamming distance processing 4 pixels, simultaneously. The minimum cost computation uses recursive XOR mechanism using AVX registers, which will reduce the number of instructions to logarithmic scale. The naïve implementation takes 2 s, where as our implementation took only 20 ms with a speedup of 100x on Intel Xeon CPU.

Table 1. Computation times of SAD/CENUS on Intel AVX

Dataset	Time in milliseconds			
	SAD		CENSUS	
	Naive	Intel AVX	Naive	Intel AVX
Cones128	2569	**164**	2215	**14**
Tsukuba32	1733	**171**	1464	**12**
Teddy128	2640	**165**	2199	**16**
Sawtooth32	2530	**159**	2193	**14**

SGM Algorithm. SGM [10] tries to minimize a global cost function with smoothness constraints using the Eq. 1. The notations are same as in the SGM [10]. The cost (C) at each pixel (p) in the disparity space (d) is updated with the penalized (P1, P2) minimum cost of the previous pixel(L') in a given direction(r).

$$L'_r(p,d) = C(p,d) + \min\begin{pmatrix} L'_r(p-r,d), \\ L'_r(p-r,d-1)+P_1, \\ L'_r(p-r,d+1)+\mathrm{P}_1, \\ \min\limits_i L'_r(p-r,i)+P_2 \end{pmatrix} \tag{1}$$

The naïve CPU implementation takes a very long time (>4 s for Teddy Image) as it involves global cost optimization. SGM has four steps, first step computes the initial cost per disparity value, second step updates the costs with eight directional optimizations and third step aggregates the cost and the fourth applies cost minimization for disparity estimation.

In the first step, SGM proposes mutual information based cost metric, but in literature, other matching cost computation methods achieved either similar or better results. In this work, we have tested various cost metrics and found census transform to be better as a distance measure. The optimization on the computation of CENSUS vector is already discussed. So we explain the cost update and aggregation steps in detail.

In 8-directional cost update step, for each direction, boundaries are identified and are filled with existing cost values from the initial cost matrix. The 8 directional cost updates are parallelized using multi-threading. In each direction, cost is updated sequentially using the SGM equation. Since the cost is an 8-bit value, SGM update equation for 16 cost values can be processed in a single AVX-128 vector instruction. The minimum of the previous pixels' disparity values is needed as per the SGM equation. The minimum is computed using recursive XOR mechanism which reduces the number of instructions to logarithmic scale.

In the cost aggregation step, all the 8 directional costs are aggregated and the accumulated cost will require 2 bytes memory. This requires processing the high and low byte separately due to which we can effectively process only eight costs at once instead of 16 as in previous step. The minima computation is done along with this step to save the memory access. This approach has a $\sim$50x speed up in case of Teddy image with 128 disparity levels on Intel Xeon E5 processor with 20-core architecture. The results with other datasets are tabulated in Table 1 and Table 2.

Table 2. Computation times of SGM on GPU

Dataset	Time in milli seconds	
	Naive	Intel AVX
Cones128	4286	**81**
Tsukuba32	2807	**54**
Teddy128	4252	**82**
Sawtooth32	4129	**78**

3.2 Nvidia Jetson Implementation

The algorithm implementation is custom designed for our target - Jetson TK1 board. The main challenges with this board are the lower number of cores (192), slow texture memory and thread availability. These challenge are addressed by exploiting the shared memory per block in the GPU, CUDA PTX instruction set, diving the summation in two steps, better memory coalescing and using CUDA shuffle instructions for accessing data from neighboring threads inside a GPU warp. We designed the implementation for maximum thread utilization to use the available cores to the full extent possible instead of maximum number of threads. This has led to a speedup of $\sim$30x which enable us to use SGM in real time applications.

SAD/CENSUS Algorithm. In both these algorithms, at every pixel, a window of size 9×7 is required, which makes it imperative to use 2D GPU blocks. Images are divided into GPU bocks of size 32×32 with each thread working on one pixel. At every pixel, 63 global memory read operations are required. To reduce this, texture memory is generally used for images but in Jetson GPU, texture memory implementation is actually found to be slower than global memory, which can be attributed to limited resources in Jetson. So shared memory is used in this work to reduce the memory read operations and results are shown in Table 3. Every GPU block has a fixed amount of shared memory and all thread in the block use this shared memory. Padded blocks of size $(32 + 8) \times (32 + 6)$ are used to manage pixels on borders as shown in Fig. 2. Each thread populates the shared memory by single read from the global memory. So the total number of global memory reads per thread are reduced from 63 to $(40 * 38)/(32 * 32) \sim= 1.5$.

Table 3. CENSUS vector computation times

Time (msec)	Global memory	Texture memory	Shared memory
Cones	5.6	7.7	**2.95**
Tsukuba	3.29	5.28	**1.86**
Teddy	6.5	8.2	**2.3**
Sawtooth	5.22	7.47	**2.82**

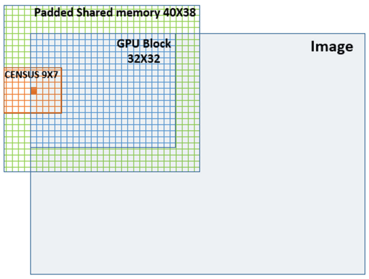

Fig. 2. GPU Kernel design of the 2D blocks for cost computation

In case of SAD algorithm, a single GPU kernel computes the disparity from the two images. The absolute differences are computed using CUDA PTX Instruction and are saved in the shared memory. Summation of this absolute difference in a window can be used for neighboring window summation. This is done using thread synchronization inside a GPU block in two steps, one for all vertical summations in the shared memory and then horizontal summation for each pixel. This saves considerable compute time.

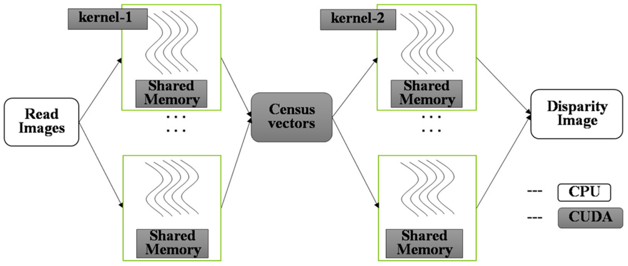

Fig. 3. GPU block diagram for CENSUS

In case of CENSUS, two kernels are used as shown in Fig. 3. In the first kernel, shared memory is used to create CENSUS vectors at every pixel in both images. This kernel is very fast since only comparison operation is used. In the second kernel, the hamming distance is computed for each pixel in left image with all candidates in the right image. Since the candidates are horizontal epipolar line, each row in the image is processed by one GPU block as shown in Fig. 4(a) and the full row of the right image is saved in the shared memory. Global memory reads per thread are reduced from N (disparity range) to 2 (one per image). In case of large images, shared memory may not be enough to save the full row. In such cases, data can be processed in multiple chunks. For better memory coalescing, each GPU block uses Disparity Range (N) number of threads. The minimum computation is also done in this hamming distance kernel and the best disparity is written to the disparity image.

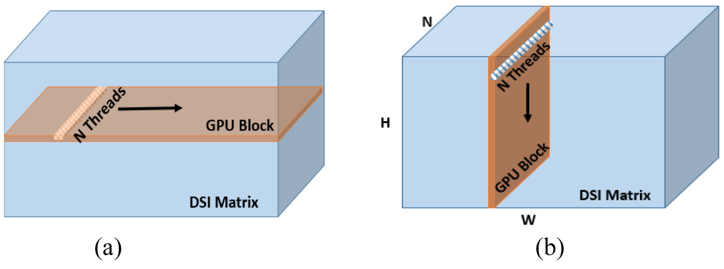

Fig. 4. GPU kernel design for (a) Hamming distance, and (b) for cost aggregation in top-down direction

SGM Algorithm. SGM has 4 main steps; cost computation, cost aggregation, cost minimization and post processing are implemented using multiple kernels. The cost computation is same as CENSUS kernel already discussed. The hamming kernel is similar except that instead of finding minimum, the actual cost values are saved to a 3D cost matrix. The cost matrix is designed for better memory coalescing by placing all the disparity levels for a pixel in contiguous memory locations.

In the 8-directional cost update step, for each direction, a kernel is designed and these kernels can run in parallel basing on streaming microprocessor (SMP) availability on the GPU. Each kernel updates the cost on separate cost 3D matrix. This enables us to have one write operation in each thread i.e. fast compute at the expense of more memory requirement. For top to bottom direction kernel, we can see that there is vertical dependency of data. So every vertical column is processed by a GPU Block as shown in Fig. 4(b). Shared memory cannot be used in this case as the dependency for each row is the previous row which will be varying at every iteration. Inside each block, numbers of threads is chosen to be equal to the disparity range (N), which is 128 in our case. We have designed to use 32 threads in a single block and each thread processing 4 disparity values, thus working on 128 disparity levels in total. Special CUDA shuffle instructions are used for communication across threads inside a warp (32 threads) to compute minimum. The description is demonstrated in the Fig. 5 where a 8 lanes are shown and in each step half the comparisons are done to get to the minimum. Each of these lanes is a thread.

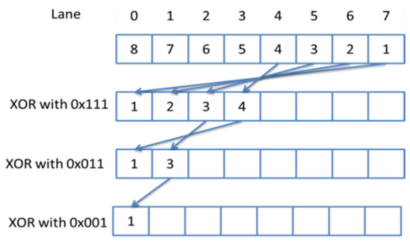

Fig. 5. XOR minima computation

The other directions of left to right, right-left and bottom-top also follow a similar approach, with only the direction and orientation of GPU block varying. In the case of diagonal kernels, slanted image slices are used. At boundaries, the column index is wrapped back to zero so that it starts at the first column of the next row. This way, only W blocks are required and each block works on same number of pixels for diagonal kernels too.

The cost aggregation and cost minimization happens in the last cost update kernel itself, which saves one memory access for every pixel since current cost is already with the thread. Minimum computation is done using CUDA shuffle instructions and CUDA PTX, which improves the efficiency. To optimize further, vectorization is used to process four disparity values in a single instruction of 4 bytes.

To suppress salt and pepper noise, median filter is applied to the disparity image using the 2D tiled GPU blocks. Occlusion is removed by using left to right consistency check. This can be performed on the same aggregated cost matrix instead of computing again from right image to left image. To manage discontinuities, background aware interpolation is used. By using these post-processing steps, accuracy is improved further but the speed reduces and should be used only if very accurate depth is required. The results are tabulated in Table 4 and Table 5.

Table 4. Computation times of SAD/CENUS on GPU

Time (msec)	SAD		CENSUS	
	Naive	Jetson	Naive	Jetson
Cones	2569	**30**	2215	**12.4**
Tsukuba	1733	**25**	1464	**10**
Teddy	2640	**31**	2199	**11.4**
Sawtooth	2530	**31**	2193	**11.8**

Table 5. Computation times of SGM on GPU

Time (msec)	Naive	Jetson
Cones	4286	**122**
Tsukuba	2807	**71**
Teddy	4252	**130**
Sawtooth32	4129	123

The computational complexities of SAD algorithm is O(WxHxDxN) where W, H are width and height of the image, D is the maximum disparity levels and N is the window size. For naïve CENUS algorithm the complexity is the same as SAD, whereas two pass CENSUS algorithm has a complexity O(WxHx(D+N)). We can see that the multiplication (NxD) part has been divided into 2 steps which reduced the complexity to addition (N+D). Since SGM uses the 2 pass CENSUS feature vectors, the complexity of SGM is given as O(WxHx(N+18D)) where the cost aggregation and minima computation leads to the 18 times access of each element in the DSI. Given a 640 $\times$ 480 image, with window size of 9 $\times$ 7 and 128 disparity values the complexities are 2.5 $\times$ 109, 5.8 $\times$ 107, 7.2 $\times$ 108 respectively for SAD, 2-pass CENSUS and SGM. We can notice that the computational complexity of SGM is less than SAD but 3D array memory access slows down the SGM drastically in practical implementation.

Middleburry dataset have been used for testing the algorithms and the qualitative bad pixel percentages are tabulated in Table 6. This paper shows that the computation time is optimized for the exact same result as the naïve implementation and so the bad

Table 6. Qualitative comparison of SAD, CENSUS and SGM

Error pixels (%)	SAD	CENSUS	SGM
Cones	18.3	20.1	6.9
Tsukuba	14.0	12.2	5.7
Teddy	16.6	18.2	7.1
Sawtooth	19.2	20.1	7.0

pixel numbers will not increase and remain exactly same for both implementations. The algorithms have also been tested on real-time ZED stereo camera images as shown in Fig. 6. As the ground truth is not available for these images, accuracy cannot be computed.

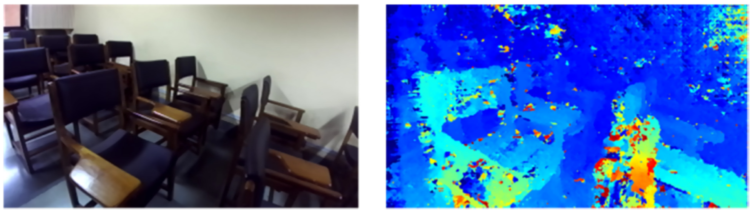

Fig. 6. Left image and SGM disparity image

4 Conclusion

In this work, it is established that optimum utilization of resources led real time implementation of stereo vision algorithms which was otherwise not feasible. Optimization of the algorithm on Intel Xeon CPU is shown to achieve $\sim$50x speed-up without using any extra hardware, compared to naïve implementation on same. Also efficient design of parallelization framework on Nvidia Jetson GPU is proven to increase the speed by 30x compared with naïve implementation on the Intel Xeon CPU with < 10 W power usage. Several techniques for optimization in CPU as well as GPU are discussed which led to the improvement, such as OpenMP multi-threading, two pass approach, vectorization, recursive XOR for minima computation, AVX register usage, usage of shared GPU memory, CUDA shuffle instructions and better memory coalescing. We achieved real time implementation of these algorithms and demonstrated the results on recorded stereo data as well as on the live images from ZED stereo camera using Nvidia Jetson board as well as Intel Xeon processor. Using these depth images for fast obstacle avoidance on a drone is the immediate extension of this work.

References

1. Mur-Artal, R., Tardós, J.D.: ORB-SLAM2: an open-source slam system for monocular, stereo, and RGB-D cameras. IEEE Trans. Rob. **33**(5), 1255–1262 (2017)
2. Broggi, A., Caraffi, C., Fedriga, R.I., Grisleri, P.: Obstacle detection with stereo vision for off-road vehicle navigation. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005)-Workshops (2005)
3. Anderson, C.S., Zhang, J., Cornea, M.: Enhanced vector math support on the Intel® AVX-512 architecture. In: 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH) (2018)
4. Fusiello, A., Trucco, E., Verri, A.: A compact algorithm for rectification of stereo pairs. Mach. Vis. Appl. **12**(1), 16–22 (2000). https://doi.org/10.1007/s001380050120
5. Mattoccia, S., Tombari, F., Di Stefano, L.: Fast full-search equivalent template matching by enhanced bounded correlation. IEEE Trans. Image Process. **17**(4), 528–538 (2008)
6. Banks, J., Bennamoun, M., Corke, P.: Non-parametric techniques for fast and robust stereo matching. In: TENCON 1997 Brisbane-Australia, Proceedings of IEEE TENCON 1997, IEEE Region 10 Annual Conference, Speech and Image Technologies for Computing and Telecommunications (Cat. No. 97CH36162) (1997)
7. Kanade, T., Okutomi, M.: A stereo matching algorithm with an adaptive window: theory and experiment. IEEE Trans. Pattern Anal. Mach. Intell. **16**(9), 920–932 (1994)
8. Fusiello, A., Roberto, V., Trucco, E.: Efficient stereo with multiple windowing. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (1997)
9. Bergen, J.R., Anandan, P., Hanna, K.J., Hingorani, R.: Hierarchical model-based motion estimation. In: Sandini, G. (ed.) ECCV 1992. LNCS, vol. 588, pp. 237–252. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55426-2_27
10. Hirschmuller, H.: Stereo processing by semiglobal matching and mutual information. IEEE Trans. Pattern Anal. Mach. Intell. **30**(2), 328–341 (2007)
11. Kolmogorov, V., Zabih, R.: Computing visual correspondence with occlusions using graph cuts. In: Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001 (2001)
12. Mayer, N., et al.: A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2016)
13. Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A., Brox, T.: FlowNet 2.0: evolution of optical flow estimation with deep networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
14. Zhong, Y., Dai, Y., Li, H.: Self-supervised learning for stereo matching with self-improving ability. arXiv preprint arXiv:1709.00930 (2017)
15. Fan, R., Dahnoun, N.: Real-time implementation of stereo vision based on optimised normalised cross-correlation and propagated search range on a GPU. In: 2017 IEEE International Conference on Imaging Systems and Techniques (IST) (2017)
16. Honegger, D., Oleynikova, H., Pollefeys, M.: Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (2014)
17. Hernandez-Juarez, D., Chacón, A., Espinosa, A., Vázquez, D., Moure, J.C., López, A.M.: Embedded real-time stereo estimation via semi-global matching on the GPU. Procedia Comput. Sci. **80**, 143–153 (2016)
18. Cui, H., Dahnoun, N.: Real-time stereo vision implementation on Nvidia Jetson TX2. In: 2019 8th Mediterranean Conference on Embedded Computing (MECO). IEEE (2019)