Near real-time Fast Bilateral Stereo on the GPU

Stefano Mattoccia
DEIS - ARCES
University of Bologna
stefano.mattoccia@unibo.it

Marco Viti
DEIS
University of Bologna
vitimar@gmail.com

Florian Ries
ARCES
University of Bologna
fries@arces.unibo.it

Abstract

State of the art local stereo correspondence algorithms that adapt their supports to image content allow to infer very accurate disparity maps often comparable to algorithms based on global disparity optimization methods. However, despite their effectiveness, accurate local approaches based on this methodology are also computationally expensive and several simplifications aimed at reducing their computational load have been proposed. Unfortunately, compared to the original approaches, the effectiveness of most of these simplified techniques is significantly reduced. In this paper, we consider an efficient and accurate algorithm referred to as Fast Bilateral Stereo (FBS) that enables to efficiently obtain results comparable to state of the art local approaches describing its mapping on GPUs with CUDA. Experimental results on two NVIDIA GPUs show that our CUDA implementation delivers, on standard stereo pairs, accurate and dense disparity maps in near real-time achieving speedup greater than 100X with respect to the equivalent CPU-based implementation.

1. Introduction

Due to its relevance in several practical applications (*e.g.* 3D reconstruction, robot vision, object recognition and categorization, surveillance and many others) inferring 3D information from standard imaging systems is a relevant topic in computer vision. Stereo vision uses a pair of synchronized cameras sensing the same scene from different viewpoints. If the *intrinsic* and *extrinsic* parameters [14] of the stereo system are computed by means of *calibration*, solving the *correspondence problem* (i.e. finding the projections of the same point of the scene into the two image of the stereo camera) allows to obtain depth by means of triangulation. Given a point in one of the two images, the geometry of a stereo camera restricts the search area in the other image to a 1D domain. Therefore, although not mandatory, in most cases stereo pairs are *rectified* [14] so as to have corresponding points that lie on the same image scanline. In this paper, as usual in this area, we assume we are dealing with rectified stereo pairs.

The correspondence problem is a crucial and difficult task extensively reviewed in [14] and [13]. Scharstein and Szeliski classify most stereo algorithms in two major classes: *local* algorithms and *global* algorithms. Moreover, according to this taxonomy, most algorithms perform some of the following four phases: *cost computation*, *cost aggregation*, *disparity computation* and *disparity refinement*. Global algorithms typically ignore cost aggregation focusing on the disparity optimization phase by means of efficient energy minimizations techniques [15] based on Belief Propagation or Graph Cut. On the other hand, local approaches typically ignore disparity optimization focusing on the cost aggregation phase performed on a small patch around each point referred to as the *support*. Extensive reviews and evaluation of state of the art local algorithms can be found in [17, 18]. According to the standard Middlebury evaluation website [12] in most cases local algorithms are outperformed by global algorithms. Nevertheless, state of the art local algorithms that perform cost aggregation by means of an *adapting weight* strategy obtain accuracy comparable to global ones. Unfortunately, and similarly to global algorithms, most of the approaches based on adapting weight strategies are computationally expensive and often not suited for practical applications.

In this paper we consider a local algorithm, referred to as Fast Bilateral Stereo (FBS) [6], based on a framework for cost aggregation that enables to obtain much more efficiently results comparable to state of the art local stereo algorithms based on adapting weights. The FBS algorithm on a CPU, despite its effectiveness compared to algorithms with similar accuracy, is not able to deliver disparity maps with frame rates suitable for most practical applications. Therefore, in this paper we show that, deploying the powerful parallel capability available in modern GPUs, the FBS algorithm can be significantly accelerated. Experimental results with a medium-class GPU show that our mapping delivers accurate and dense disparity maps in near real-time.

This paper is organized as follows. In section 2 we re-

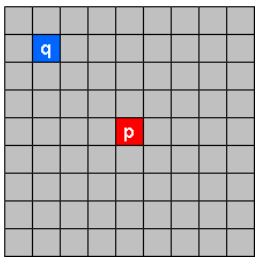

Figure 1. Adapting weigh strategies: the support S for reference or target images concerned with the central point depicted in red. The weight of point q is computed, according to different cues, with respect to point p at the center of the support.

view related work concerning state of the art local stereo algorithms based on adapting weight approaches and relevant GPU-based stereo vision algorithms. In Section 3 we propose an efficient parallel mapping of the FBS algorithm on GPUs with CUDA. Detailed experimental results provided in Section 4 highlight the notable speed-up achieved by our GPU mapping compared to a CPU-based implementation of FBS. Conclusions are drawn in section 5.

2. Related work

In this section we review state of the art local algorithms based on adapting weight approaches and relevant implementations of stereo vision algorithms on GPUs. Extensive reviews and evaluations of state of the art stereo vision algorithms can be found in [14], [13, 12] while detailed reviews and evaluations of state of the art cost aggregation strategies typically deployed by local stereo algorithms can be found in [18] and [17, 16].

2.1. Local stereo vision algorithms based on adapting weight strategies

According to [18], [17, 16] local stereo algorithms that aggregate costs by means of adapting weight strategies outperform approaches that explicitly modify the shape of their supports. The idea to assign weights according to image content to points within the supports was proposed in [10], [2] and [21]. However, the effectiveness of this idea became clear with the Adaptive Weights (AW) approach proposed by Yoon and Kweon [23]. In AW, each point (*e.g.* q in Figure 1) within the support of the reference and the target image of the stereo pair is weighted according to two strategies. The first related to the Euclidean distance $\Delta(q)_s$, referred to as *proximity* distance, between the examined point q and the point p at the center of the support. Thus, points closer to the center of the support receive higher weights. The second strategy aims at aggregating points with similar disparity. To this aim, with the disparity being unknown a priori, the cue deployed is the *color* distance. That is, this method assigns to q a weight according to the Euclidean distance $\Delta(q)_c$ between colors of p and q in defined color space (*i.e.* CIELab in AW). Therefore, points with lower $\Delta(q)_c$ are assumed more likely to be at the same disparity of the point at the center of the support. Weights are computed in the reference and target image (according to the considered disparity d). In AW, being γ_s and γ_c two parameters empirically set, the overall weight assigned to a point q within the support region of the reference ($l = R$) or the target image ($l = T$) centered in p is computed by means of:

$$W_l(p,q) = e^{\frac{-\Delta(q)_s}{\gamma_s}} \times e^{\frac{-\Delta(q)_c}{\gamma_c}} \quad (1)$$

Given a pointwise matching cost $m(q,d)$ (*e.g.*, Truncated Absolute Differences (TAD) in [23]) for point q at disparity d, the overall weighted matching cost $M(p,d)$ assigned to the support S centered in p at disparity d is:

$$M(p,d) = \frac{\sum_{q_i \in S(p)} W_R(p,q_i) \cdot W_T(p,q_i) \cdot m(q_i,d)}{\sum_{q_i \in S(p)} W_R(p,q_i) \cdot W_T(p,q_i)} \quad (2)$$

As reported in [23] and in [18, 17, 16], this symmetric strategy for cost aggregation is very effective. Nevertheless, improvements to the original adapting weight technique [23] were proposed discarding the proximity distance and using as cues segmentation, Segment Support [17], or *geodesic weights* [4]. Unfortunately, on standard stereo pairs, these techniques share with AW execution times of minutes on standard CPUs. This fact limits their deployment in most practical applications.

Therefore, techniques aimed at speeding-up the execution time of state of the art techniques for cost aggregation were proposed. In some cases (*e.g.* [19]) the computational load is reduced by computing weights asymmetrically only on the reference image. Although this choice enabled significant speed-ups, the accuracy of the resulting disparity maps is also significantly reduced.

A different approach for accelerating the AW cost aggregation strategies without affecting its effectiveness was proposed in [6]. The Fast Bilateral Stereo (FBS) algorithm combines the effectiveness of standard *correlative* approaches with the effectiveness of approaches based on the adapting weight methodology. The FBS algorithm relies on a framework that computes symmetric and simplified weights on a block basis and computes matching costs (*i.e.* TAD), precisely and in constant time, on a point basis by means of integral images [1] or box-filtering [7] techniques. The key idea behind FBS can be explained observing Figure 2. Given a support S of size $B \times B$, FBS splits S in non overlapping blocks of size $b \times b$. For each point within a $b \times b$

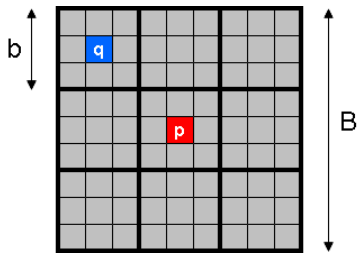

Figure 2. Fast Bilateral Stereo: the support S for reference and target images concerned with the central point p depicted in red. FBS partitions the support of size $B \times B$ in non overlapping blocks of equal size $b \times b$. Approximated proximity and color distances are computed on a block basis while matching cost is computed precisely and in constant time on a point basis.

block, a single proximity weight, computed according to the Euclidean distance between p and the center of the block q, is assigned. Moreover, to each $b \times b$ block is assigned a single color proximity distance computed according to the Euclidean distance between the color intensity of p and the average color intensity computed on the considered $b \times b$ block. The color space used by FBS is RGB. Conversely, since to each $b \times b$ block is assigned a single overall weight with a methodology similar to 2, the matching cost for each point belonging to the same block can computed precisely and efficiently in constant time by means of [1]. Moreover, the same techniques can be used to compute efficiently and in constant time the average intensity of each $b \times b$ block. This framework enables to reduce, by a factor $b \times b$, the number of weight computation 1 as well as the number of operations required to compute 2. According to the results reported in [6] and [16] FBS obtains result comparable to AW much more efficiently (according to [6], on the same standard stereo pairs, FBS requires seconds while AW requires minutes). The FBS algorithm obtained optimal results with blocks of size $b = 3$. However, parameter b can be used to trim accuracy vs efficiency [6].

2.2. Stereo vision algorithms on GPUs

Thanks to their effectiveness, GPUs are becoming a very popular computing platform in computer vision and some GPU-based stereo vision implementations have been proposed so far. In [3] Hernst and Hirschmuller implemented the Semi-Global Matching method reaching peak of 13 fps on a NVIDIA GeForce 8800 Ultra with 320×240 stereo pairs and disparity range of 64 pixels.

In [11] Richardt et al[1] proposed a real-time stereo matching technique inspired by the Yoon and Kweon's adaptive support weights algorithm [23]. Their algorithms computes weights symmetrically deploying the *bilateral grid* approach to achieve a speed-up of 200X compared to a straightforward full-kernel GPU implementation. However, they provide results on greyscale images and the accuracy of the resulting disparity maps is not equivalent to [23].

Wang et al [20] proposed a stereo algorithm that combines Dynamic-Programming with simplified cost aggregation step based on a simplified and asymmetric version of [23]. They achieves high quality results in real-time achieving over 50 million disparity evaluations per second (MDE/s).

In [22], Yang et al proposed the mapping on a GPU of a belief propagation based algorithm that generates high quality results. They reports a speed-up of 45x compared to the CPU equivalent implementation.

Kalarot and Morris, in [8] compared the performance of the same stereo vision algorithm on FPGA and and GPU so as to better understand the advantages of each high performance computing architecture.

3. Fast Bilateral Stereo on the GPU

In this section we describe the mapping of the FBS algorithm on GPUs with CUDA [9, 5]. Our GPU implementation was evaluated on a medium-class NVIDIA GeForce GTX 460 card and a high-end NVIDIA Tesla C2070. The GTX 460 has 7 multiprocessors, each one with 48 processors with a clock of 1350 Mhz, and has 1 GB of global memory GDDR5 operating on a memory clock of 1800 Mhz. The Tesla C2070 has 14 multiprocessors, each one with 32 processors with a clock of 1150 Mhz, and has 6 GB of global memory GDDR5 operating on a memory clock of 1500 Mhz.

3.1. CUDA: Compute Unified Device Architecture

Nvidia's Compute Unified Device Architecture (CUDA) provides a generic driver architecture for GPUs; it offers a flexible programming model with minimal extensions to C, allowing the programmer to define C function, called *kernels*, that, when called, are executed within the GPU in parallel by different *threads*.

In CUDA a GPU is seen as made of many *Streaming Multiprocessors* ($SM_0 \ldots SM_{n-1}$), each SM consists of a set of Streaming Processors ($SP_0 \ldots SP_{m-1}$). Each multiprocessor has on-chip memory of four types: one set of local *registers* per scalar processor, one memory shared by its scalar processors called *shared memory*), two read-only caches called *constant memory* and *texture memory*, respectively. Moreover, all multiprocessors have access to the same *global memory*. Finally, each multiprocessor has one - or two, depending on the GPU's Compute Capability - common instruction unit which, thanks to a very lightweight

[1]This is an updated version of the paper. The originally published paper unintentionally failed to describe properly this approach.In fact, this algorithm computes weights symmetrically and its loss in accuracy, compared to the adaptive weights method, comes primarily from the use of greyscale images. We apologize for this unintentional mistake.

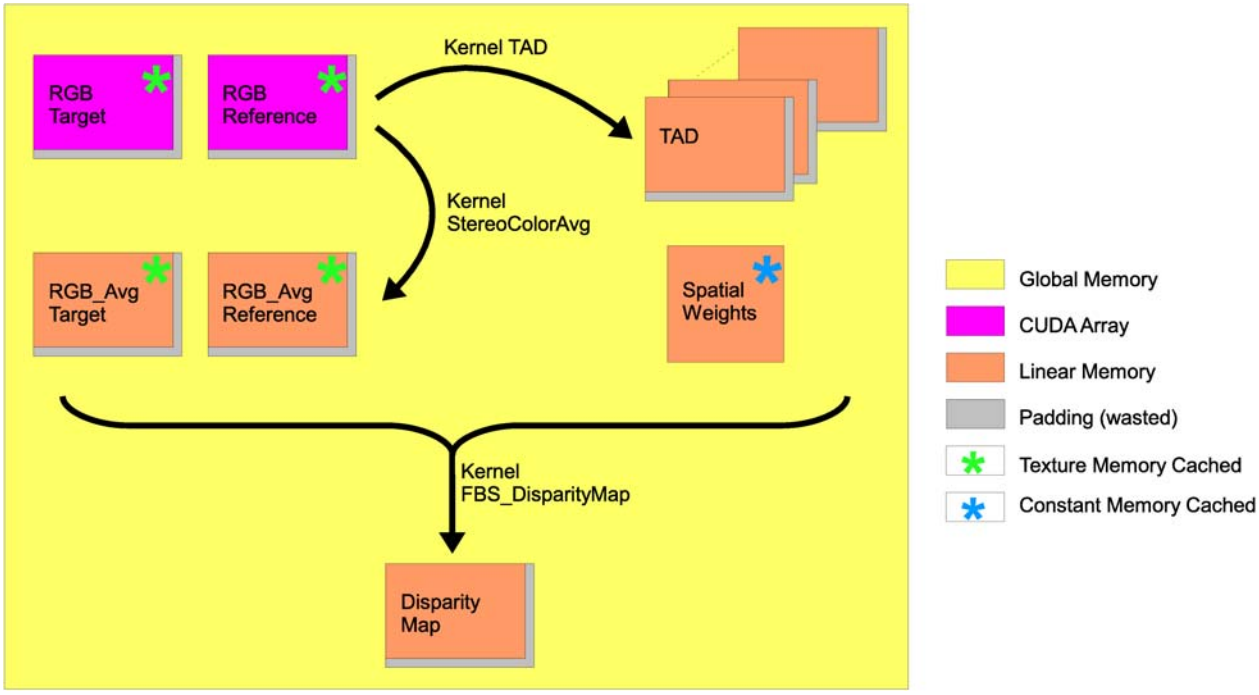

Figure 3. Different types of memory deployed by the three kernels that map the FBS algorithm on the GPU.

thread scheduling mechanism, creates, manages and executes concurrent threads with zero scheduling overhead.

A number k of threads ($t_0 \ldots t_{k-1}$) can be assigned to an SM forming a thread *block*. A block physically maps to an SM but the number of threads in a block can be greater than number of SPs; anyway, threads are split into $warps$ - each one composed of 32 threads - and only one warp is active at a given time. Active threads execute the same instruction in parallel. If there is a branch in the code, threads agreeing on instructions execute in parallel while the other branches wait for them to complete.

Performance optimization revolves around three basic strategies: *(a)* maximize parallel execution to achieve maximum utilization; *(b)* optimize memory usage to achieve maximum memory throughput; *(c)* optimize instruction usage to achieve maximum instruction throughput. Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion: for example, optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain.

3.2. Parallel mapping of the FBS algorithm on the GPU

In this paper we exploit the intrinsic parallelism exposed by the GPU assigning to each thread one pixel of the reference image. For each pixel the thread will process the entire disparity range δ. We split the entire FBS algorithm in 3 main portions, each one associated with a kernel.

- The first kernel, referred to as *StereoColorAvg*, takes as input the reference and target images in order to compute the average RGB values for the reference and target images required by FBS to compute the weights for each block $b \times b$.
- The second kernel, referred to as *TAD*, takes as input the reference and target images and computes the matching cost (*i.e.* TAD for FBS) of the pixels within each block of size $b \times b$.
- The third kernel referred to as *FBS_DisparityMap* computes, for each pixel of reference image, the weighted cost of each possible correspondence with the pixels of the target within the disparity range δ. Once these operations are completed, the kernel searches for the candidate with the minimum cost within the disparity range δ.

With typical parameters [6] of the FBS algorithm (*i.e.* with $b \leq 7$ and $B \geq 39$), the FBS_DisparityMap kernel, compared to the other two kernels, it is the most computationally expensive on GPUs. Therefore we initially evaluated the occupancy of each kernel changing the number of threads for each block in order to maximize the occupancy of the the FBS_DisparityMap kernel. This preliminary study highlighted that, with typical parameters of

the FBS algorithm, the optimal occupancy can be obtained with blocks of 256 threads. With this block size we observed an occupancy of 0.667 during the execution of the FBS_DisparityMap kernel and an occupancy of 0,833 during the execution of the other two kernels. When the size of the stereo pair is not multiple of the the block size (*i.e.* 256), the input images are increased to satisfy this constrain (*padding*) so as to avoid the expensive handling of out of memory memory accesses. In the experimental result section we report (Table 2) the average execution time of each module of the GPU implementation of the FBS algorithm.

It is worth noting that the first two kernels are independent of each other, thus can run simultaneously on the GPU. On the other hand, the third kernel requires results provided by the first two kernels; therefore, it can start only when the other two kernels have completed their execution. Figure 3 shows in detail how the different types of memory available in the GPU are used by the three kernels. Finally, since a major bottleneck in our GPU implementation of FBS arises from the frequent use of arithmetic instructions with low throughput (*i.e.* exponentials and square roots), we used intrinsics rather than regular functions for cost computation on the GPU. Intrinsics are less accurate but the difference in the final disparity map are in most cases negligible.

3.2.1 StereoColorAvg and TAD kernels

The texture cache is optimized for 2D spatial locality, so threads of the same warp that read close texture addresses in 2D will achieve the best performance. Therefore, in order to reduce the access time to reference and target images, we used texture memory for StereoColorAvg and TAD kernels. Moreover, we store the stereo pair as 2D CUDA array being this data structure read only and optimized for texture fetching. The pseudo-codes in Figure 4 describe, respectively, the operations executed by the StereoColorAvg and TAD kernels.

3.2.2 FBS_DisparityMap kernel

Once completed the execution of the StereoColorAvg and TAD kernels, and before launching the third and final FBS_DisparityMap kernel, some preliminary operations are required. In order to reduce the global memory latency, it is important to bind the texture memory, as well as the reference and target images, with the average RGB values - already computed by the StereoColorAvg kernel. Similarly to the strategy adopted for reference and target images, average RGB values should be stored in CUDA arrays - therefore copied from the read-write linear memory to CUDA array. However, we observed that the overhead associated with the copy makes more efficient to leave average RGB values in linear memory.

```
each thread do:
{
  determine which is
    the pixel to process (x,y);
  compute the sums of RGB values
    of the pixels within the blocks
    centered in (x,y), for the
    Reference and for the Target;
  normalize the two sums;
  write these values in global memory;
}

each thread do:
{
  determine which is
    the pixel to process (x,y);
  for each disparity d
  {
    compute the TAD of RGB values of
      the pixels within the block
      centered in (x,y) in the Reference
      and the block centered in (x-d,y)
      in the Target;
    write this value in global memory;
  }
}
```

Figure 4. Pseudo-code for (Top)the StereoColorAvg kernel and (Bottom) the TAD kernel.

The second step aims at avoiding redundant computations concerned with the proximity distance weights required by each thread. Therefore, these weights are computed only once by the CPU, copied into the global memory of the GPU and shared by the threads. Finally, these weights are bound to the constant memory cache so as to speed-up accesses. It is noteworthy that when a warp does a constant memory request, it is split into as many separate requests as there are different memory addresses in the request, decreasing throughput by a factor equal to the number of separate requests. Fortunately, synchronizing the threads, the spatial weights requests are always at the same address, so the maximum throughput is achieved. The pseudo-code reported in Figure 5 describes the operations executed by the FBS_DisparityMap kernel.

4. Experimental results

In this section we provide[2] a detailed evaluation of our mapping on two NVIDIA GPUs. We compare the FBS algorithm proposed in [6] to the implementation of the same algorithm on the GPU. Both versions of FBS include sub-

[2]Additional experimental results are available at this web page: www.vision.deis.unibo.it/smatt/FBS_GPU.html

```
each thread do:
{
  determine which is
    the pixel to process (x,y);
  for each disparity d
  {
    compute and save the cost M at d;
    compare the cost M with the best one;
    update min disparity and cost;
  }
  scale min disparity;
  write this value in global memory;
}
```

Figure 5. Pseudo-code of the FBS_DisparityMap kernel.

pixel disparity estimation by means of a second degree fitting in proximity of the best score found by the algorithm. In Figure 6 we report the disparity maps computed by the proposed GPU implementation of the FBS algorithm, the CPU implementation of the FBS algorithm and the result of the AW [23] algorithm available, in [17, 16]. Detailed errors and disparity maps, computed on the Middlebury dataset [13, 12], for the FBS and AW algorithms are available in [16]. The Figure confirms, qualitatively, that our GPU mapping provides disparity maps equivalent to those provided by our CPU implementation. It is worth noting that, although the algorithms mapped on the CPU and on the GPU are almost equivalent, the results reported in Figure 6 show slight differences. These differences mostly occur at the edges of the images (*e.g.* within the yellow box depicted in Figure 6), due to the different handling of out-of-range texture coordinates in the GPU, and in low textured regions (*e.g.* within the green circles depicted in Figure 6) due to the numerical approximations of the intrinsic functions deployed. Nevertheless, our quantitative comparison confirms that in most points the results obtained with the GPU version FBS algorithm are equivalent to those obtained by the original CPU version. For our experiments we used the same parameters and configurations of the FBS algorithms reported in [6].

Table 1 shows the performance speed-up achieved by our mapping of the FBS algorithm on a medium-class NVIDIA GeForce GTX 460 and on a high-end NVIDIA Tesla C2070 with respect to our CPU implementation of the FBS algorithm on a CPU (using a single core). For both cases we report results according to the Middlebury dataset with two different configurations of the FBS algorithms: with parameters $B = 39, b = 3$ and with parameters $B = 45, b = 5$. Table 1 clearly highlights the notable speed-up achieved by the proposed GPU implementation in both examined configurations. Using the GTX 460, for the larger stereo pairs Teddy and Cones, with disparity range $\delta = 60$, the speed-up is greater than 70X for $B = 39, b = 3$ and greater than 60X for $B = 45, b = 5$ corresponding, respectively, to 3.3 and 5.5 fps. On the Tsukuba stereo pair the frame rate is 15.3 and 25 for $B = 39, b = 3$ and $B = 45, b = 5$, respectively. With the high-end Tesla C2070 we were able to improve the performance of about 30% obtaining speed-ups greater than 100X with parameters of the FBS algorithms $B = 39, b = 3$ and around 80X on average with parameters $B = 45, b = 5$. It is worth to note that the Tesla is optimized for double precision floating point operations that we do not use in our implementation of the FBS algorithm on GPU. Moreover the Tesla C2070, compared to the GTX 460, has also a higher bandwidth and a larger global memory. These latter features would be more effective dealing with stereo pairs larger than those used for our experiments. The experiments with both devices confirm the effectiveness of the proposed GPU mapping of the FBS algorithm on GPUs.

Stereo Pair	GPU	Execution Time [msec]		Measured Speed–up	
		(a)	(b)	(a)	(b)
Tsukuba	GTX 460	65	40	65,5X	53,2X
384×288	C2070	46	29	101X	75,9X
$\delta = 16$					
Venus	GTX 460	114	69	71,7X	57,4X
434×383	C2070	80	51	103X	82,3X
$\delta = 20$					
Teddy	GTX 460	302	178	72,7X	60,1X
450×375	C2070	201	122	105X	87,4X
$\delta = 60$					
Cones	GTX 460	303	181	72,4X	60,6X
450×375	C2070	200	122	106X	87,1X
$\delta = 60$					

Table 1. Results of columns *(a)* are obtained with a supports of size 39×39 and block of size 3×3, while the ones of columns *(b)* are obtained with a supports of size 45×45 and block of size 5×5. The GPUs used for these experiments were an NVIDIA GeForce GTX 460 and an NVIDIA Tesla C2070. The CPU was an Intel Celeron E3300 @ 2,50 GHz (using a single core).

In Table 2 we also report the average *breakdown* of the GPU processing time for the two previously considered configurations of the FBS algorithm. The table reports that most of the time is spent executing the FBS_DisparityMap kernel. Nevertheless, it is worth noting that, increasing the size of the block size b the percentage of time spent executing the TAD kernels, compared to the overall execution time, gets more relevant . This behavior can be explained observing that with larger b the number of accesses to the texture memory increases reducing, due to worse data locality, the texture cache hit rate. The execution time of the TAD kernel increases proportionally to b. Nevertheless,

with block size $b = 3$ and $b = 5$, typically used by FBS this problem is almost negligible.

Operation	Time Used [%]	
	(a)	(b)
Host to GPU memcpy	0,25	0,49
StereoColorAvg	0,10	0,20
TAD	3,48	16,28
FBS_DisparityMap	95,98	82,63
GPU to Host memcpy	0,19	0,40

Table 2. Breakdown of GPU time: these values are obtained, on a GTX 460, averaging the breakdowns on the Tsukuba, Venus, Teddy and Cones stereo pairs. Results reported in column *(a)* are concerned with supports of size 39×39 and block of size 3×3, while results reported in column *(b)* are concerned with supports of size 45×45 and block of size 5×5.

5. Conclusions

In this paper we have proposed the mapping of Fast Bilateral Stereo, an efficient algorithm with accuracy comparable to state of the art approaches based on adapting weight cost aggregation strategies, on a GPU with CUDA. Our proposal enables, on standard stereo pairs, to deliver disparity maps in near-real time. The measured speed-up, with respect to an efficient implementation of the FBS algorithm on CPU, is greater than 70X on a GTX 460 GPU and greater than 100X on a Tesla C2070 GPU.

Acknowledgements

The authors would like to thank NVIDIA for the donation of the Tesla C2070 GPU.

References

[1] F. Crow. Summed-area tables for texture mapping. *Computer Graphics*, 18(3):207–212, 1984. 137, 138

[2] T. Darrel. A radial cumulative similarity transform for robust image correspondence. In *Proc. Conf. on Computer Vision and Pattern Recognition*, pages 656–662, 1998. 137

[3] I. Ernst and H. Hirschmüller. Mutual information based semi-global stereo matching on the gpu. In *Proceedings of the 4th International Symposium on Advances in Visual Computing*, ISVC '08, pages 228–239, 2008. 138

[4] A. Hosni, M. Bleyer, M. Gelautz, and C. Rhemann. Local stereo matching using geodesic support weights. In *ICIP*, 2009. 137

[5] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010. 138

[6] S. Mattoccia, S. Giardino, and A. Gambini. Accurate and efficient cost aggregation strategy for stereo correspondence based on approximated joint bilateral filtering. In *Proc. of ACCV2009*, 2009. 136, 137, 138, 139, 140, 141, 143

[7] M. Mc Donnel. Box-filtering techniques. *Computer Graphics and Image Processing*, 17:65–70, 1981. 137

[8] J. Morris and R. Kalaot. Comparison of fpga and gpu implementation of real-time stereo vision. In *ECVW 2010*, 2010. 138

[9] NVIDIA. Nvidia. www.nvidia.com. 138

[10] K. Prazdny. Detection of binocular disparities. *Biological Cybern*, 52:9399, 1985. 137

[11] C. Richardt, D. Orr, I. Davies, A. Criminisi, and N. A. Dodgson. Real-time spatiotemporal stereo matching using the dual-cross-bilateral grid. In *ECCV (3)*, pages 510–523, 2010. 138

[12] D. Scharstein and R. Szeliski. Middlebury stereo vision. http://vision.middlebury.edu/stereo/. 136, 137, 141

[13] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. Jour. Computer Vision*, 47(1/2/3):7–42, 2002. 136, 137, 141

[14] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010. 136, 137

[15] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Trans.PAMI*, 30(6):1068–1080, 2008. 136

[16] F. Tombari, S. Mattoccia, L. Di Stefano, and E. Addimanda. Classification and evaluation of cost aggregation methods for stereo correspondence. www.vision.deis.unibo.it/spe/SPEHome.asp. 137, 138, 141, 143

[17] F. Tombari, S. Mattoccia, L. Di Stefano, and E. Addimanda. Classification and evaluation of cost aggregation methods for stereo correspondence. In *CVPR08*, pages 1–8, 2008. 136, 137, 141, 143

[18] L. Wang, M. Gong, M. Gong, and R. Yang. How far can we go with local optimization in real-time stereo matching. In *Proc. Third Int. Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT 2006)*, pages 129–136, 2006. 136, 137

[19] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3DPVT '06*, pages 798–805, 2006. 137

[20] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *Proc. 3rd Int. Symposium 3D Data Processing, Visualization and Transmission (3DPVT'06)*, pages 798–805, 2006. 138

[21] Y. Xu, D. Wang, T. Feng, and H. Shum. Stereo computation using radial adaptive windows. In *Int. Conf. on Pattern Recognition*, volume 3, pages 595– 598, 2002. 137

[22] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nistér. Real-time global stereo matching using hierarchical belief propagation. In *BMVC*, pages 989–998, 2006. 138

[23] K. Yoon and I. Kweon. Adaptive support-weight approach for correspondence search. *IEEE Trans. PAMI*, 28(4):650–656, 2006. 137, 138, 141

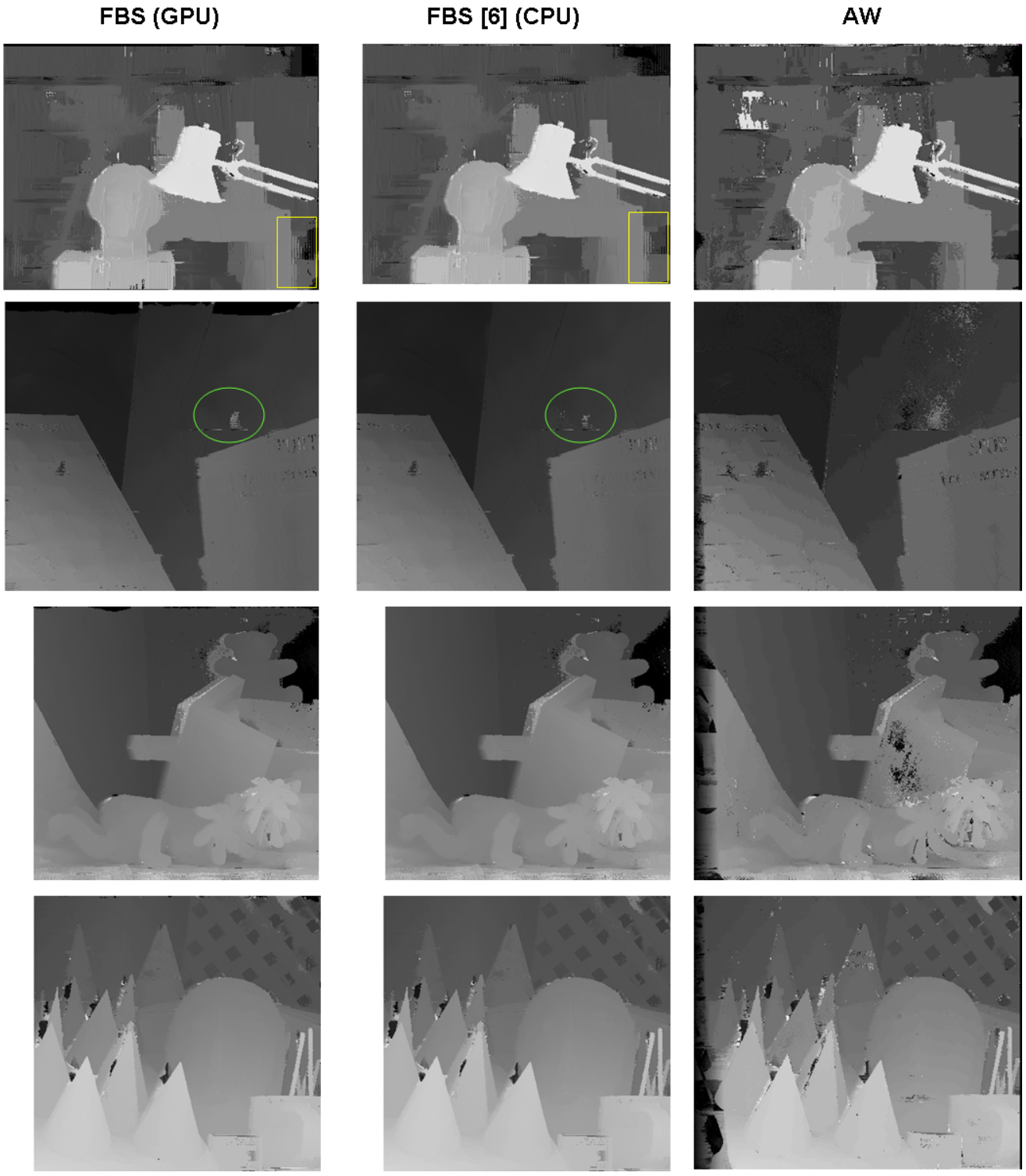

Figure 6. Disparity maps concerned with the images of the Middlebury dataset. (Left) Results of the proposed implementation of FBS on GPU (Center) Results of the FBS algorithm [6] on a CPU (Right) Results of the Adaptive Weights (AW) algorithms according to [17, 16].