Image and Vision Computing 29 (2011) 420–432

Contents lists available at ScienceDirect

Image and Vision Computing

journal homepage: www.elsevier.com/locate/imavis

# Real-time stereo on GPGPU using progressive multi-resolution adaptive windows ☆

Yong Zhao *, Gabriel Taubin

*Division of Engineering, Brown University, 182 Hope Street, Providence, RI 02912, United States*

ARTICLE INFO

*Article history:*
Received 28 December 2009
Received in revised form 7 December 2010
Accepted 27 January 2011

*Keywords:*
Real-time stereo
GPGPU

ABSTRACT

We introduce a new GPGPU-based real-time dense stereo matching algorithm. The algorithm is based on a progressive multi-resolution pipeline which includes background modeling and dense matching with adaptive windows. For applications in which only moving objects are of interest, this approach effectively reduces the overall computation cost quite significantly, and preserves the high definition details. Running on an off-the-shelf commodity graphics card, our implementation achieves a 36 fps stereo matching on 1024×768 stereo video with a fine 256 pixel disparity range. This is effectively same as 7200 M disparity evaluations per second. For scenes where the static background assumption holds, our approach outperforms all published alternative algorithms in terms of the speed performance, by a large margin. We envision a number of potential applications such as real-time motion capture, as well as tracking, recognition and identification of moving objects in multi-camera networks.



## 1. Introduction

Estimating depth from stereo is a classical computer vision problem, which has received significant attention since the early days. Recovering 3D information from a pair of stereo camera has been a popular topic because the additional 3D information provided by this technology contains significantly more information than 2D information produced by traditional cameras. Some believe that this technology will fundamentally revolutionize the computer vision signal processing pipeline, as well as how future cameras will be built.

However, this 2D to 3D evolution has always been facing many challenges, which can be grouped into two main categories: accuracy and efficiency. Accuracy becomes an important concern in applications such as precise 3D surface modeling, especially when dealing with object surfaces with complex reflectance behavior, rich geometric structure, significant amount of occlusion and poor texture. Efficiency is the main concern when the stereo system is employed in real-time applications such as robot navigation, video surveillance, and interactive user interfaces.

Unfortunately these challenges often conflict with each other: in order to improve the quality of stereo matching, people usually cast the problem as a global optimization problem, which results in high computation cost and poor efficiency. On the other hand, most efficient stereo matching algorithms are based on only local information, which leads to poor accuracy under some difficult situations.

The focus of our work is to provide real-time high resolution stereo for applications in which only foreground moving objects are of interest, such as motion capture, object tracking, recognition and identification in a visual sensor network (VSN) scenario. These applications usually have some special performance constraints: first, being able to estimate depth in real-time speed is essential for the whole system to be able to work in real-time; second, high resolution is very important for applications which have large working volumes, and very crowded scenes, where attention to detail information is necessary; third, fine scanning range (maximum disparity) is necessary for applications which have to deal with big depth of view; after all, stereo is usually only one part of a real-time computer vision processing pipeline. In order to have the whole pipeline works in real-time, a faster-than-real-time stereo system is needed to save significant amount of processing time (both CPU time and GPU time) for other higher level processing tasks. In the following sections, we will explain in detail how this daunting task is accomplished with commodity computer graphic hardware.

## 2. Related work

### 2.1. Vision signal processing using GPGPU

Nowadays, the performance of modern graphic hardware has reached the point where billions of 3D model elements can be rendered in real-time. The main reason for this achievement is an architecture composed of a huge array of simple processors, which are efficiently organized. Besides 3D rendering, this massive parallel computing resource can actually be used to speed-up many other applications, including computer vision signal processing.

☆ This paper has been recommended for acceptance by Vladimir Pavlovic.

* Corresponding author. Tel.: +1 4018636179.
*E-mail addresses:* yongzhao@brown.edu (Y. Zhao), taubin@brown.edu (G. Taubin).

0262-8856/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.imavis.2011.01.007

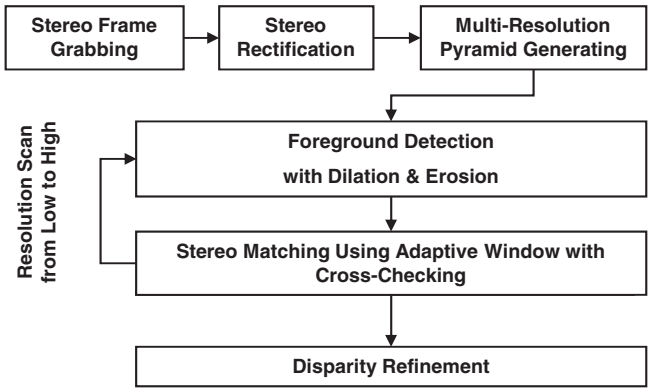


**Fig. 1.** Processing pipeline of our stereo algorithm.

Regardless of the short history of General Purpose GPU (GPGPU) technology, many computer vision researchers are among the first group of pioneers who have explored the general use of GPU parallel computing resource on various kinds of computer vision tasks, such as local feature detection [1], edge detection [2], and stereo matching [3–9], to name a few.

In this paper, we have chosen to use Nvidia graphics cards, and their Compute Unified Device Architecture (CUDA) programming model, to process our stereo data. Since even a brief introduction of CUDA would be too large to be presented here, we refer interested readers to [10] for detailed architecture information.

### 2.2. Real-time stereo matching algorithms

During the past 20 years, many stereo systems have been successfully developed with high flexibility, compact size and acceptable cost. The overall stereo matching literature is too large to be surveyed here. We refer interested readers to [11] for a taxonomy and evaluation. Instead, in this section we briefly mention recent progress in GPU-based real-time stereo algorithms.

In 2003, Yang and Pollefeys presented a multi-resolution real-time stereo algorithm implemented in a commodity graphics hardware platform [12]. The traditional sum-of-square-difference (SSD) was used to independently aggregate matching cost on different resolutions. The final matching cost was determined here as the sum of the matching costs from the different resolutions. A winner-take-all scheme was used to determine the disparity. This multi-resolution scheme effectively reduced noise from the high resolution matching errors and provided a smoother depth map. However, the accuracy of the depth map, especially on the image area where depth discontinuities take place, was compromised. They achieved 50–70 M disparity evaluations per second (MDE/s), resulting in a quasi-real-time system which, at that time, set a new record for stereo matching speed.

In 2004, Yang and Pollefeys presented a new GPU-based adaptive window approach [13]. Instead of the traditional fixed-size square window, their cost aggregation windows can adaptively change shape according to the content of the local image area, taking into account edges and corners. This work successfully pushed the stereo matching

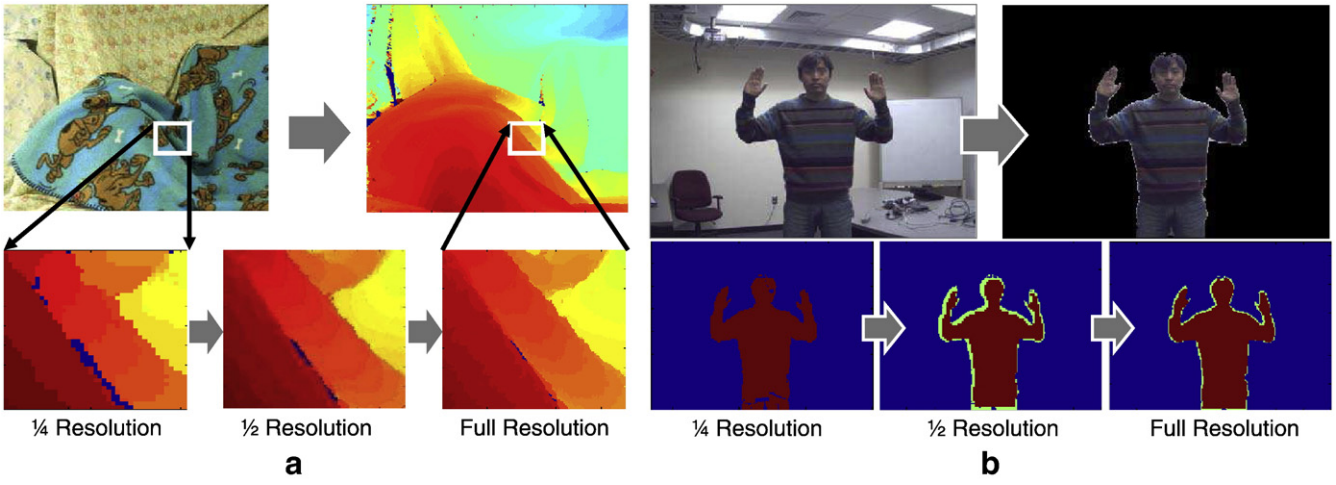


**Fig. 2.** (a) Coarse-to-fine matching on multiple resolutions; (b) foreground detection is performed on multiple resolutions, from low to high. On higher resolutions, the actual foreground detections only takes place at the boundary blocks of foreground blobs. Only the green pixels on the above foreground masks represent the foreground pixels detected by the pixel-wise comparison to the background model.

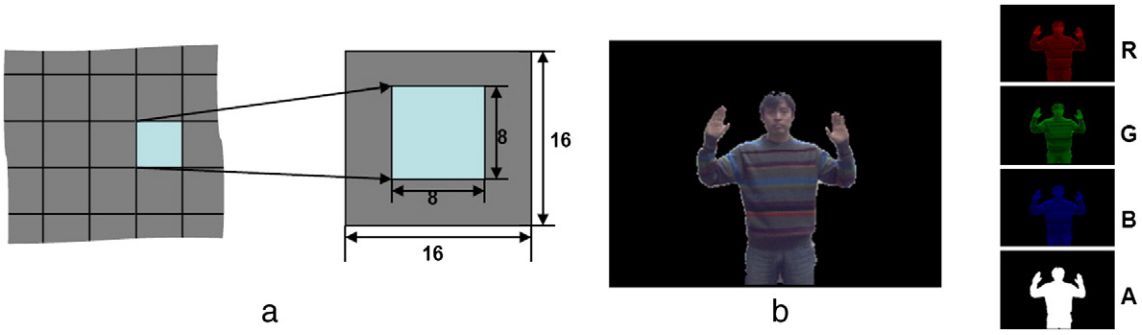


**Fig. 3.** (a) CUDA implementation of foreground detection; (b) the foreground mask is packed with the color image at the A channel of RGBA formatted image.

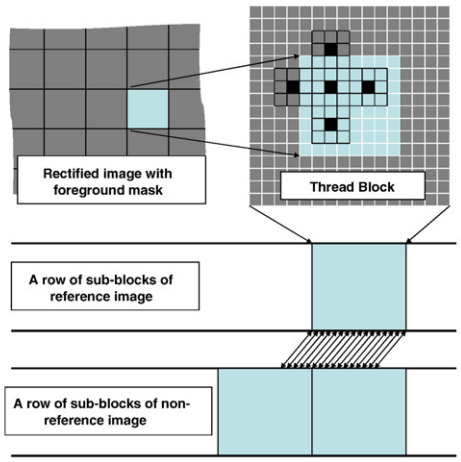


**Fig. 4.** CUDA implementation of dense matching using adaptive window.

speed record up to 289 MDE/s using more modern computer graphics hardware.

More recently, a number of new GPGPU-based stereo matching algorithms have been presented [3–9]. Some of them emphasize different features, such as wide dynamic range for outdoor use, or semi-global optimization for better accuracy. It is worth to mention the work of Yang et al. [9], in which a belief propagation based global algorithm is managed to run at real-time on GPU. It might be the first time that the high quality which is usually only available from those slow global algorithms, was achieved in real-time.

## 3. Method

### 3.1. Processing pipeline

Fig. 1 shows the pipeline of our algorithm. Stereo video frames are captured and rectified, so that a pair of row-aligned images can be obtained to reduce the computation cost of matching. However, the stereo matching doesn't start from the original input resolution. Instead, a pyramid of down-sampled images are computed from the rectified pair, denoted by $\{I_1, I_2, \cdots, I_N\}$. $N$ is the number of total resolutions. $I_N$ is original full resolution of stereo frame, and each $I_i$ is the half sample of $I_{i+1}$. Before stereo matching starts, multi-resolution background modeling on both view, denoted by $\{B_1, B_2, \cdots, B_N\}$ is applied on stereo views to detect the multi-resolution foreground mask $\{F_1, F_2, \cdots, F_N\}$. The foreground detection result is used to improve the matching performance. The stereo dense matching will be processed on the lowest resolution $I_1$ first, then progressively on other higher resolutions.

During this process, the disparity searching is performed only on the searching range suggested by disparity result from lower resolution. This iteration continues until the processing on the original full resolution $I_N$ is completed. Fig. 2(a) shows some

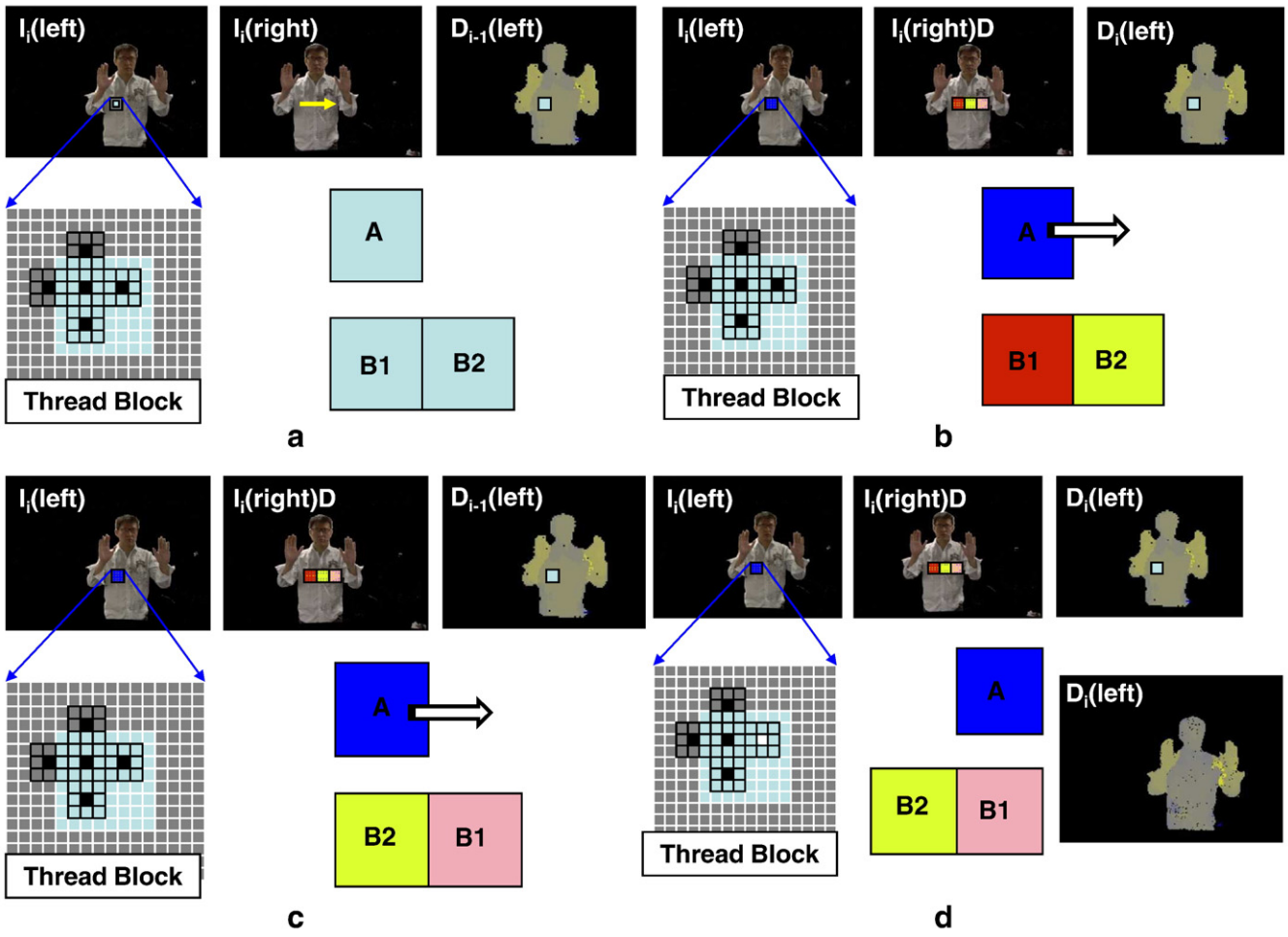


**Fig. 5.** CUDA implementation of dense matching using adaptive window.

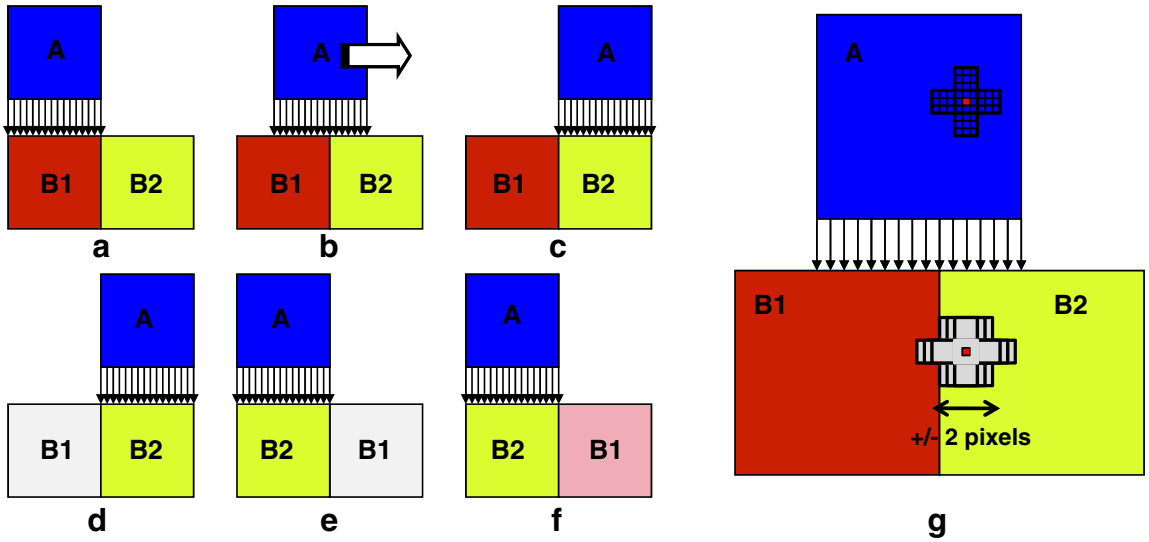


**Fig. 6.** CUDA implementation of dense matching using adaptive window. (a) Disparity sweeping starts after buffers B1 and B2 are loaded with the first and second pixel blocks. (b) During each step of disparity sweeping, each pixel of the 8×8 pixel block in the center part of buffer A, is compared with the pixels in the corresponding region in B1–B2 combined buffer. After each step, a one-pixel offset is applied on the horizontal direction. (c) The disparity sweeping keeps proceeding till it reaches the end of the B1–B2 combined buffer. (d) Buffer B1 is cleared. (e) A new B2–B1 combination is formed. (f) Next 16×16 pixel block from right view is loaded in buffer B1, then the disparity sweeping keeps going.

intermediate results from this coarse-to-fine multi-resolution process. Once the original resolution disparity map is computed, sub-pixel disparity refinement is optionally applied in applications that need better precision.

### 3.2. Multi-resolution background modeling

The foreground segmentation is embedded in our stereo matching pipeline. Because of many applications, only moving objects are of interest. In these cases, the uninteresting background part of the image results not only in a waste of processing time, but also in additional sources of matching errors.

Our foreground detection algorithm is based on the algorithm introduced by K. Jain et al. [14], which is an efficient variation of Gaussian background modeling [15]. This background modeling is applied on HSV space instead of RGB space in order to reduce shadow effects. This background modeling algorithm works fine with most indoor scenarios and is computational economic enough for real-time applications. However, any other background modeling algorithms can be used in our stereo matching pipeline as long as they can produce a binary mask indicating a foreground or background decision.

In our multi-resolution pipeline, the foreground detection algorithm is applied on the pyramid, progressively from low to high resolution. In addition, two extra steps are taken to accelerate the processing on multiple resolutions and improve the smoothness of foreground detection, respectively.

First, stereo images on different resolutions are divided into square blocks in the foreground detection. For each pixel block from $I_i$ ($i>1$), the foreground detection results $F_{i-1}$ from the lower resolution $I_{i-1}$, are checked. If all the pixels of the corresponding block in $I_{i-1}$ have uniform detection results (either all foreground or all background), then this result will be copied to all the pixels of the block on $I_i$. Otherwise, pixel-wise comparison between $I_i$ and $B_i$ is applied to decide whether it is a foreground pixel or not. This process is illustrated in Fig. 2(b). By doing this, only pixels of the foreground boundary blocks are actually tested against the background model. At the same time, most noises on foreground blobs at high resolutions are removed. In our practice, this technique can effectively reduce the total number of pixels which need to be tested by up to 90%.

Second, foreground dilation and erosion are applied on the foreground detection result, which fills the holes inside foreground blobs. After this, a pyramid of nice and smooth foreground detections are obtained.

The CUDA implementation of this multi-resolution foreground detection algorithm comprises multiple passes, each of which works on a different resolution. In each pass, the steps of background update, foreground detection, foreground dilation, and foreground erosion are all performed by one CUDA kernel function. As illustrated in Fig. 3(a), each 16×16 thread block takes care of loading and processing a 16×16 pixel square block. However only the center 8×8 pixel square is saved. This mapping gives each 8×8 pixel block a 4 pixel wide "apron" which overlaps with surrounding blocks. The overlapped part of image is processed redundantly so that the foreground dilation and erosion would not cause artifacts at the block boundary area. The

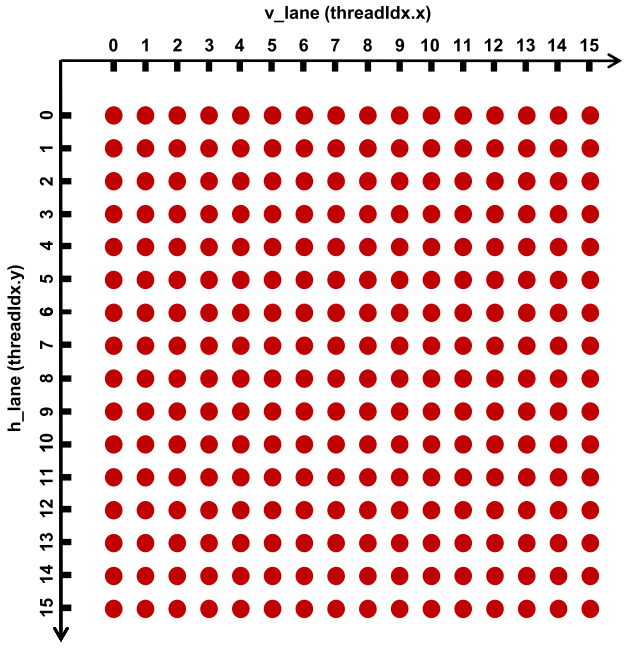


**Fig. 7.** Integrate a 16×16 2D image block in a CUDA thread block.

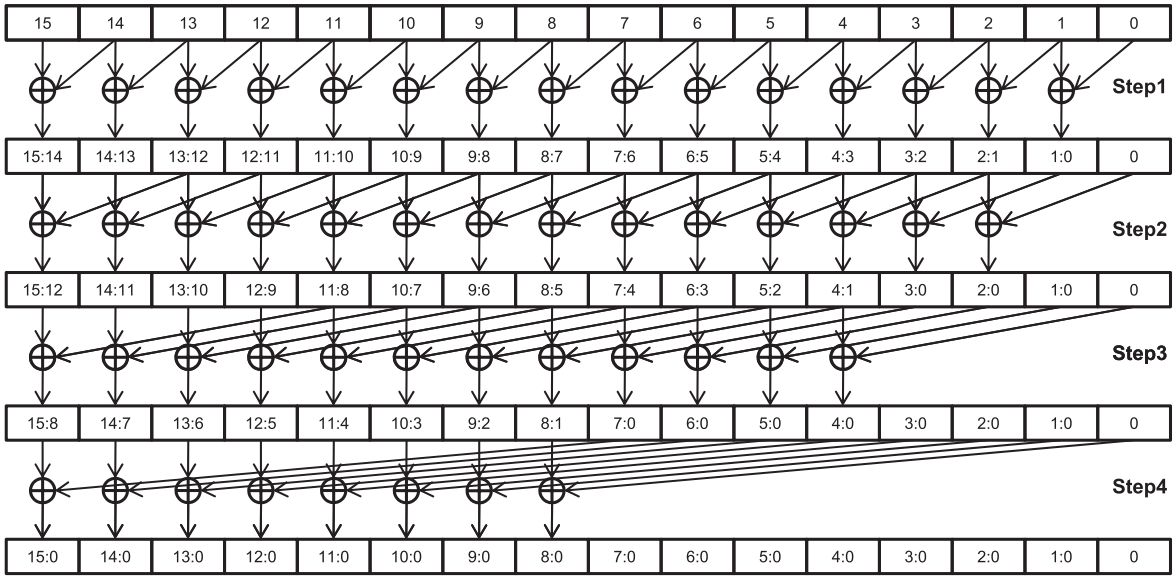


**Fig. 8.** Integrate a 1D pixel array with 16 pixels by a CUDA half-warp.

result of foreground detection is saved in the Alpha channel of the rectified input images, which are represented in RGBA format, as indicated in Fig. 3(b).

### 3.3. Multi-resolution stereo matching

In our algorithm, stereo matching starts from the lowest resolution. The adaptive window approach introduced by Yang and Pollefeys [13] is adopted here to compute the matching cost. The reason is that this approach results in higher matching accuracy compared to the fixed window approaches, given the same number of involved neighboring pixels. However, as the resolution of image and range of disparity scan increases, the $4\times4$ sub-window configuration in this algorithm eventually becomes insufficient. This is why this approach is used only on the lowest resolution $I_1$.

For example, we need to compute disparity map for a stereo pair with full resolution $W_N\times H_N$ and disparity searching range $S_N$. The lowest resolution of this stereo input is $\frac{W_N}{2^{N-1}}\times\frac{H_N}{2^{N-1}}$, and the corresponding disparity searching range $S_1=\frac{S_N}{2^{N-1}}$. For each pixel on the left view of the lowest resolution, denoted by $I_1(left,x,y)$ (*left* means left view of stereo pair, $x$ and $y$ are pixel coordinates on horizontal and vertical direction, respectively), the process of computing its disparity $D_1(left,x,y)$ is as follows:

- If $F_1(left,x,y)=1$, continue; otherwise, no disparity for $I_1(left,x,y)$.
- For $s\in[x,x+S_1]$, compute the matching cost $C(s)$ using adaptive window approach if $F_1(right,x+s,y)=1$. If $F_1(right,x+s,y)=0$, set the matching cost $C(s)=\infty$.
- Disparity $D_1(left,x,y)=argmin_sC(s)$, where $s\in[x,x+S_1]$.

Once the stereo matching on the lowest resolution is done, stereo matching is performed on higher resolutions progressively. For each pixel of left view at resolution $i$, denoted by $I_i(left,x,y)$, the process of computing its disparity $D_i(left,x,y)$ is as follows:

- If $F_i(left,x,y)=1$, continue; otherwise, no disparity for $I_i(left,x,y)$.
- For $s\in\left[2\cdot D_{i-1}\left(left,\frac{x}{2},\frac{y}{2}\right)-2,2\cdot D_{i-1}\left(left,\frac{x}{2},\frac{y}{2}\right)+2\right]$, compute the matching cost $C(s)$ using adaptive window approach if $F_i(right,x+s,y)=1$. If $F_i(right,x+s,y)=0$, set the matching cost $C(s)=\infty$.
- Disparity $D_i(left,x,y)=argmin_sC(s)$, where $s\in\left[2\cdot D_{i-1}\left(left,\frac{x}{2},\frac{y}{2}\right)-2,2\cdot D_{i-1}\left(left,\frac{x}{2},\frac{y}{2}\right)+2\right]$.

```
__device__ inline void integrate_2D(float* im_block, int v_lane, int h_lane)
{
   // scan along horizontal direction
   if (v_lane>=1) im_block[h_lane*16+v_lane] += im_block[h_lane*16+v_lane-1]; // step1
   if (v_lane>=2) im_block[h_lane*16+v_lane] += im_block[h_lane*16+v_lane-2]; // step2
   if (v_lane>=4) im_block[h_lane*16+v_lane] += im_block[h_lane*16+v_lane-4]; // step3
   if (v_lane>=8) im_block[h_lane*16+v_lane] += im_block[h_lane*16+v_lane-8]; // step4

   __syncthreads();

   // scan along vertical direction
   if (v_lane>=1) im_block[v_lane*16+h_lane] += im_block[v_lane*16+h_lane-1]; // step1
   if (v_lane>=2) im_block[v_lane*16+h_lane] += im_block[v_lane*16+h_lane-2]; // step2
   if (v_lane>=4) im_block[v_lane*16+h_lane] += im_block[v_lane*16+h_lane-4]; // step3
   if (v_lane>=8) im_block[v_lane*16+h_lane] += im_block[v_lane*16+h_lane-8]; // step4

   return;
}
```

**Fig. 9.** Sample code of CUDA device function to integrate a $16\times16$ 2D image block.

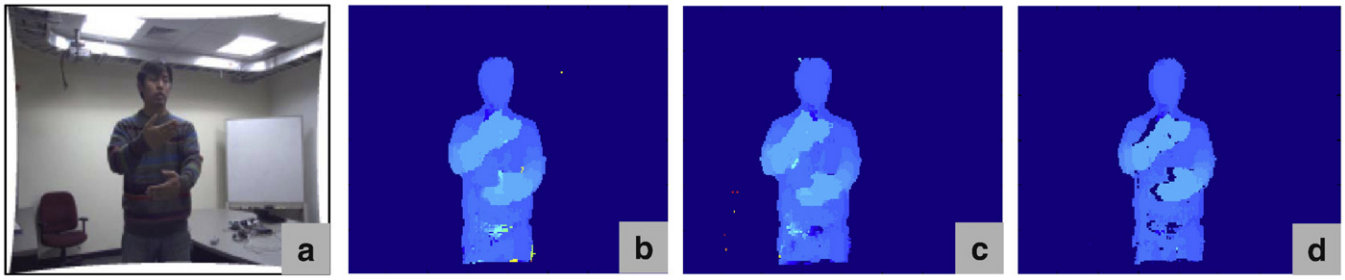


**Fig. 10.** (a) Input image; (b) disparity map from stereo matching with left view as the reference image; (c) disparity map from stereo matching with right view as the reference image; (d) depth map after cross-checking. Matching error caused by occlusion and poor texture is successfully removed.

Notice that two steps are taken to minimize the disparity searching range: 1) the foreground detection result is used in this process in such a way that, only foreground pixels from both views are considered for possible matching. 2) For each resolution$I_i$ ($i>1$), we limit the searching to a 4 pixel span centered at the suggestion from the disparity result on the lower resolution $D_{i-1}$. By doing this, the matching process is significantly accelerated, and the matching accuracy is improved as well.

Our algorithm is different from one of the most well-known multi-resolution real-time stereo approaches [12]. In that work, Yang et al. propose that for each pixel, matching cost is aggregated on a number of different resolutions, and the final disparity is decided by first

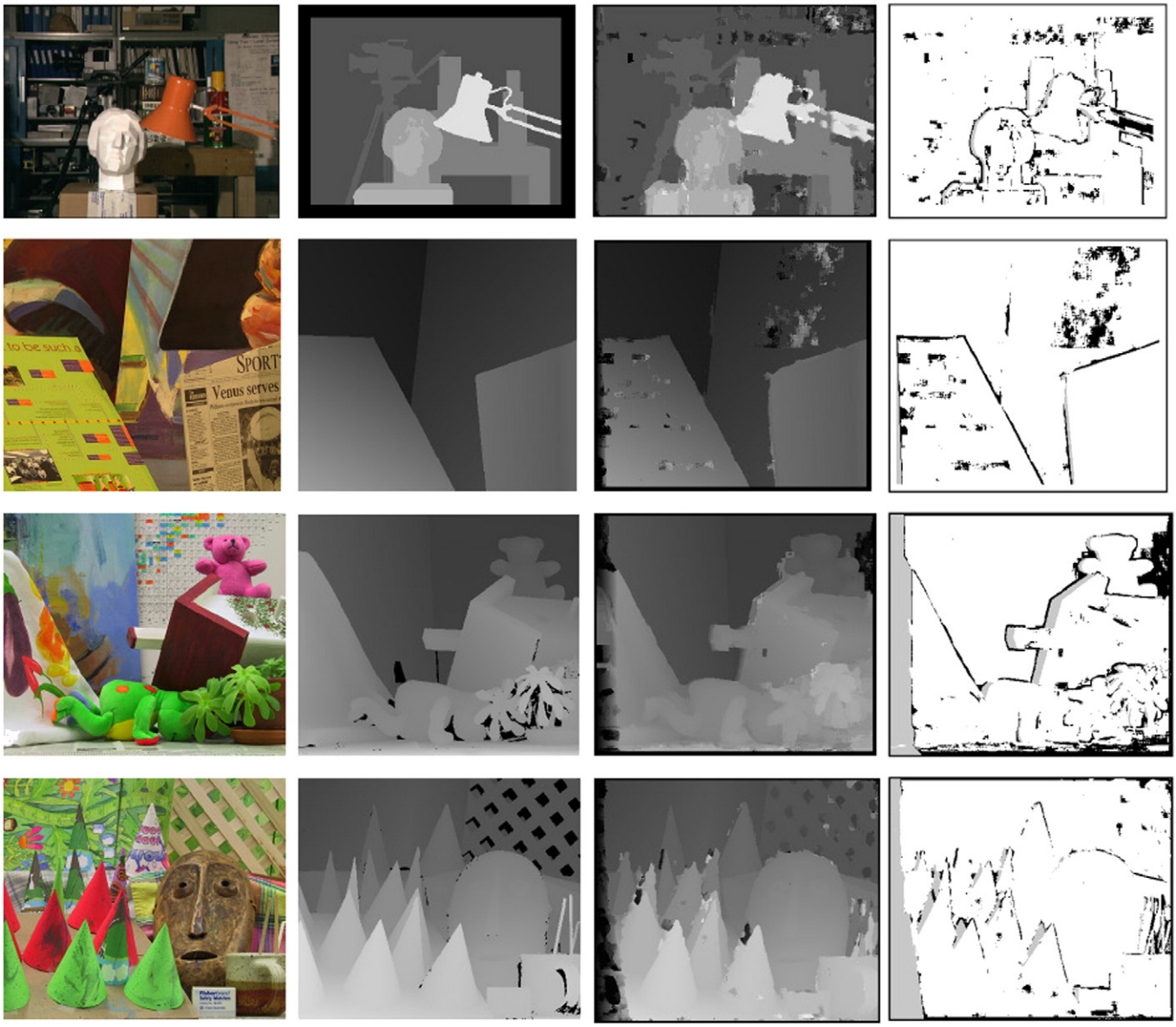

**Fig. 11.** The disparity maps of the benchmark stereo image pairs on Middlebury website obtained by our multi-resolution adaptive window algorithm. The first row — Tsukuba, the second — Venus, the third — Teddy, and the fourth — Cones. The first column — reference image, the second — ground truth, the third — our results, and the fourth — matching errors.

**Table 1**
Error rate of our method on Middlebury dataset.

| Algorithm | Avg. rank | Tsukuba | | | Venus | | | Teddy | | | Cones | | | % of bad pixels |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Nonocc | All | Disc | Nonocc | All | Disc | Nonocc | All | Disc | Nonocc | All | Disc | |
| Our method | 73.0 | 8.29 | 10.3 | 22.4 | 6.20 | 7.51 | 26.6 | 12.5 | 21.2 | 25.7 | 5.83 | 14.5 | 10.3 | 14.3 |

summing up the matching cost on all these resolutions and searching for the minimum overall cost. This basically means using the average matching costs on multiple different resolutions to decide high resolution disparity. Their approach generates good-looking disparity map, but usually lose the high resolution detail because of smoothing factor introduced by the results from lower resolutions. Also their approach computes matching cost along the full disparity searching range on all resolutions, and saves all the immediate matching cost results in global memory, which brings tremendous burden on the global memory bandwidth.

Our approach on the other hand, uses the disparity results from lower resolution to guide the searching range at higher resolution. Matching cost dose not need to be computed exhaustively on every possible test disparity along the searching range. Therefore time is saved both from computing and global memory access. And the final high resolution disparity is directly determined only by the high resolution matching cost aggregation. Therefore our approach tends to preserve high resolution details. Of course there exist chances where once a wrong decision has been made at low resolution, it can not be corrected at higher resolution. However in practice, the chance of erroneous matching at low resolution is quite low. And cross-checking can always be used to detect most low resolution matching errors. And these problematic pixels will get opportunities to be corrected on higher resolutions later.

### 3.4. Single CUDA kernel implementation of stereo matching

It's worth to mention some of our CUDA implementation details because an elaborate job of mapping processing and data to CUDA concurrent thread array (CTA) is extremely important for achieving optimal performance. As illustrated in Fig. 4, each thread block loads a $16\times16$ pixel block from the reference image but only the disparities of the center $8\times8$ pixels are evaluated. All the input data – rectified image pairs with foreground masks embedded in the Alpha channel – are bind with CUDA texture so that fast access can be achieved through a cached memory interface.

Each thread block also loads two $16\times16$ square blocks from the non-reference image. Then for each of the $8\times8$ pixels from the reference image, cost aggregation is computed using the adaptive window approach with $3\times3$ sub-window size. If searching range is bigger than 16 pixels, the thread block just loads another $16\times16$ square block from the non-reference image and repeats the cost aggregation process. The disparity evaluation finishes only after the entire searching range is covered.

Figs. 5 and 6 demonstrate how the implementation of our stereo matching algorithm at a certain resolution, using just one single kernel CUDA functions.

In Fig. 5(a), rectified stereo images (left and right view) with foreground mask at resolution $i$, as well as the left-view disparity map at resolution $i-1$, are displayed on the upper row. Shown in the lower-left of this diagram, is a CUDA thread block, or so-called the Concurrent Thread Array (CTA) taking care of estimating disparity at a $8\times8$ pixel block region. But in order to compute the matching cost using adaptive window, this CTA actually needs to load a bigger $16\times16$ pixel block which is centered at the $8\times8$ pixel block. Before the stereo matching starts, this CTA first reads the lower resolution disparity map at corresponding region shown in the upper-right. From the lower resolution disparity map, a lower and upper boundary of the disparity values of this pixel block at current higher resolution can be obtained. For example, if the lower resolution disparity map at this region varies in $[21,42]$, then at current resolution, disparity should be searched in $[42-2,84+2]$. Notice that an extra 4 pixel padding is added on both ends. On the upper-center part of this diagram, a yellow arrowed line is used to highlight the suggested disparity searching range at resolution $i$. Each CTA creates three $16\times16$ RGBA buffers in their shared memory, denoted by A, B1 and B2.

In Fig. 5(b), CTA fills three RGBA buffers A, B1 and B2 with $16\times16$ pixel blocks from left-view and right-view. Different colors are used to indicate the corresponding location: the pixel block from left-view is loaded in buffer A, the first and second pixel block along the disparity searching line are loaded in buffer B1 and B2, respectively. Once the data is ready in these fast internal buffers, a so-called "disparity sweeping" can be performed. The big white horizontal arrow on top of buffer A indicates the direction of this sweeping.

Fig. 6(a,b,c,d,e,f) illustrates the "disparity sweeping" process: Buffer A is loaded with a $16\times16$ pixel block from the left view, buffers B1 and B2 are loaded with the first and second $16\times16$ pixel blocks of right view initially. Buffers B1 and B2 are combined as a $16\times32$ pixel block. Buffer A is used to "sweep" along the combined B1–B2 buffer. At each step of this "sweeping", the matching cost of the center $8\times8$ pixel block is computed using the adaptive window approach. When sweeping reaches the end of the B1–B2 combined

**Table 2**
Accuracy and speed performance comparison of different algorithms using Middlebury dataset. The unit of speed performance measurement is Million Disparity Estimation, or MDE/s. These numbers do not necessarily mean the actual number of disparity evaluation, but the effectively equivalent speed. The accuracy performance is all evaluated using the Middlebury benchmark images. Algorithms followed by the * sign do not report Middlebury benchmark result. For these algorithms, the error rate results are from our own implementation on a PC. All speed performance numbers are from the original papers. This comparison includes the following algorithms: AdaptOvrSegBP [17], SymBP + occ [18], DoubleBP [19], EnhancedBP [20], LocallyConsist [21], CoopRegion [22], AdaptingBP [23], C-SemiGlob [24], FastAggreg [25], OptimizedDP [26], SegTreeDP [27], RealtimeVar [28], RealtimeBP [9], RealtimeGPU [29], MultiResSSD [12], PlaneFitBP [30], RealtimeDP [31], ESAW [32], ConnectivityCons [33], AdaptiveWin[13], CSBP [34], RTCensus [35], and our method.

| Algorithm name | Percentage of bad pixels | Speeds (MDE/s) | Hardware platform |
|---|---|---|---|
| AdaptOvrSegBP | 5.59 | 0.04 | 3.2 GHz CPU |
| SymBP + occ | 5.92 | 0.08 | 2.8 GHz CPU |
| DoubleBP | 4.19 | 0.1 | PC |
| EnhancedBP | 6.69 | 0.13 | PC |
| LocallyConsist | 6.33 | 0.15 | 2.5 GHz CPU |
| CoopRegion | 4.41 | 0.18 | 1.6 GHz CPU |
| AdaptingBP | 4.23 | 0.2 | 2.21 GHz Athlon 64 bit |
| C-SemiGlob | 5.76 | 0.4 | 2.8 GHz CPU |
| FastAggreg | 8.24 | 6.14 | 2.14 GHz CPU |
| OptimizedDP | 8.83 | 9.95 | 1.8 GHz CPU |
| SegTreeDP | 6.82 | 10.2 | 2.4 GHz CPU |
| RealtimeVar | 9.05 | 13.9 | 2.85 GHz CPU |
| RealtimeBP | 7.69 | 19.7 | Geforece 7900 GTX |
| RealtimeGPU | 9.82 | 53 | ATI Radeon XL1800 |
| MultiResSSD* | 17.32 | 117 | Nvidia Geforce4 |
| PlaneFitBP | 5.78 | 170 | Nvidia Geforce 8800 GTX |
| RealtimeDP | 10.7 | 187 | AthlonXP 2400+ CPU |
| ESAW | 8.2 | 194.8 | Nvidia Geforece 7900 GTX |
| ConnectivityCons* | 15.62 | 280 | Nvidia Geforce 6800 GT GPU |
| AdaptiveWin* | 19.69 | 289 | ATI Radeon 9800 |
| CSBP | 11.4 | 460 | Nvidia Geforece 8800 GTX |
| RTCensus | 9.73 | 1300 | Nvidia Geforce GTX 280 |
| Our method | 14.3 | 7200 | Nvida Geforce GTX 280 |

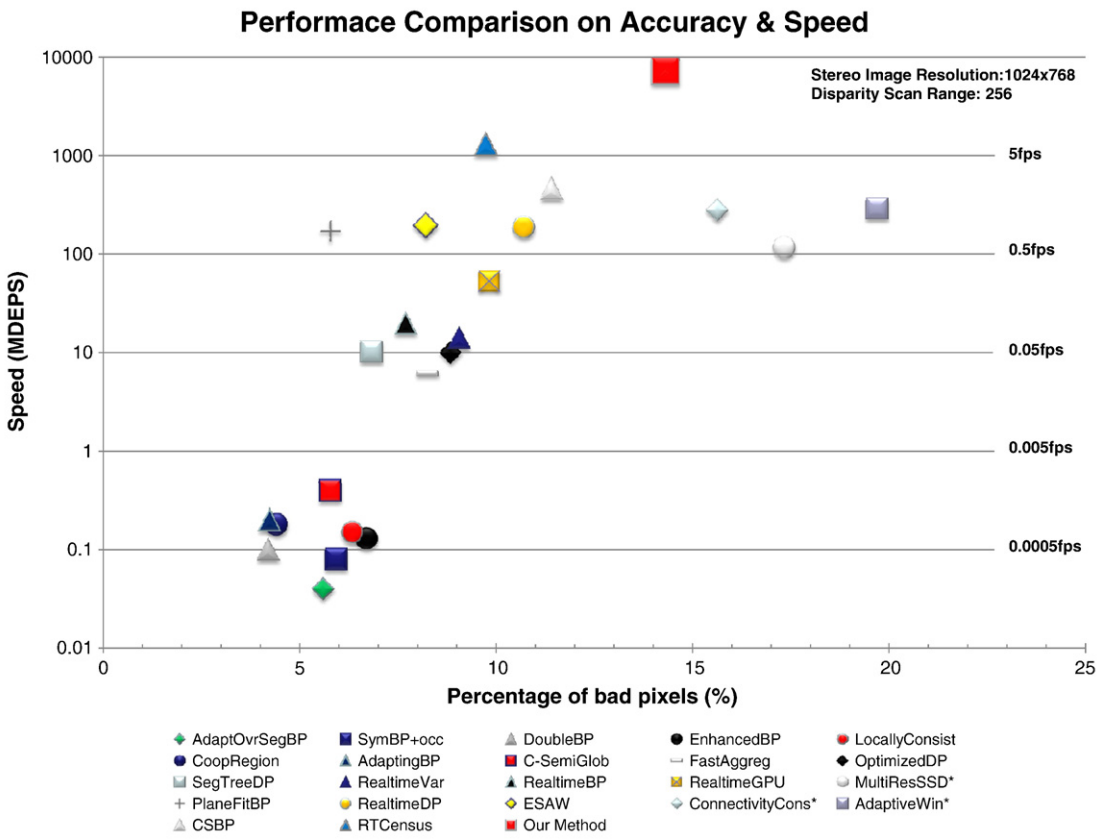


**Fig. 12.** Accuracy and speed performance comparison of different algorithms.

buffer, B1 is cleared and reloaded with the next pixel block from the right view. Then B1 is attached at the trailing end of B2 to form a B2–B1 combined buffer. Then "sweeping" can keep going till the whole disparity searching is finished.

Fig. 6(g) shows what actually happens for each pixel in the left view during the "disparity sweeping". Red color is used to highlight the pixel in buffer A, whose disparity will be estimated by the "disparity sweeping". From the corresponding location in the lower resolution disparity map, the suggested disparity of this pixel at higher resolution can be obtained, which is also highlighted using red color in the B1–B2 combined buffer. During the "disparity sweeping", each CTA travels a much longer distance. Only when the "sweeping" proceeds into a small 4 pixel range which is centered at the suggested high resolution disparity, the adaptive window matching cost will be computed for this pixel. Please notice that foreground detection information which is embedded at the Alpha channel, will also be used to constrain the matching cost computing during the "disparity sweeping" process.

Fig. 5(c) and (d) shows how "disparity sweeping" is performed on the example stereo image pair with swapping buffers B1 and B2 in the combined buffer. Finally, a left-view disparity map (shown in the lower-right of Fig. 5(d)) at resolution $i$ is created by this single CUDA kernel function.

It's worth to mention an important CUDA implementation technique which dramatically improves the speed performance of the "disparity sweeping" process — using integral image to accelerate computing matching cost in sub-windows. Integral image is a useful technique to compute the sum of values from a rectangular block efficiently. The definition of integral image is that for an image $N$, the integral image of $N$ is $I$. The value of each pixel of $I$ must satisfy: $I(x,y) = \sum_{x'=0}^{x} \sum_{y'=0}^{y} N(x',y')$. However computing integral image can be quite expensive. Here we present a very simple CUDA implementation of 2D image integration.

In Fig. 7 we show a 16×16 SAD image block stored in shared memory, which will be processed by a thread block with 16×16 threads. Each thread has a 2D thread ID (threadIdx.x, threadIdx.y). And each pixel has a 2D coordinate (h_lane, v_lane). In this example, we want to compute an integral image of this SAD image so that later we can efficiently compute the matching cost of each sub-window. We use a two-pass approach to integrate the SAD image: first integrate each row, then integrate each column.

In Fig. 8, we illustrate the process of integrate a 1D pixel array with 16 pixels using 16 threads from the same half-warp of a CUDA thread block. It takes 4 steps to finish this 1D integration. During each step, some thread replaces the value of a pixel using the sum of this pixel with the value of another pixel. Because all the threads from the same half-warp are automatically synchronized, each step can be executed simultaneously without using an extra buffer. It doesn't take a "__syncthreads()" command to synchronize the whole thread block until the horizontal integration is finished. And then we proceed with the vertical integration using the same 4 steps. A sample CUDA device function of this 2D integration is shown in Fig. 9.

Compared to Yang's GPU implementation in [13], the most remarkable improvement of our implementation is that, for each pixel, the entire disparity searching takes place within one kernel function and only the final disparity result is stored in global memory just once. This helps to speed up the processing quite significantly because writing to global memory in the GPGPU is one of the most time-consuming operations, due to the fact that there is no outbound caching in current GPGPU architecture.

### 3.5. Cross-checking

Cross-checking was firstly introduced by Cochran et al. [16]. The idea is as follows: there are two views with a pair-wise stereo. One is called reference view and the other is called target view. The stereo

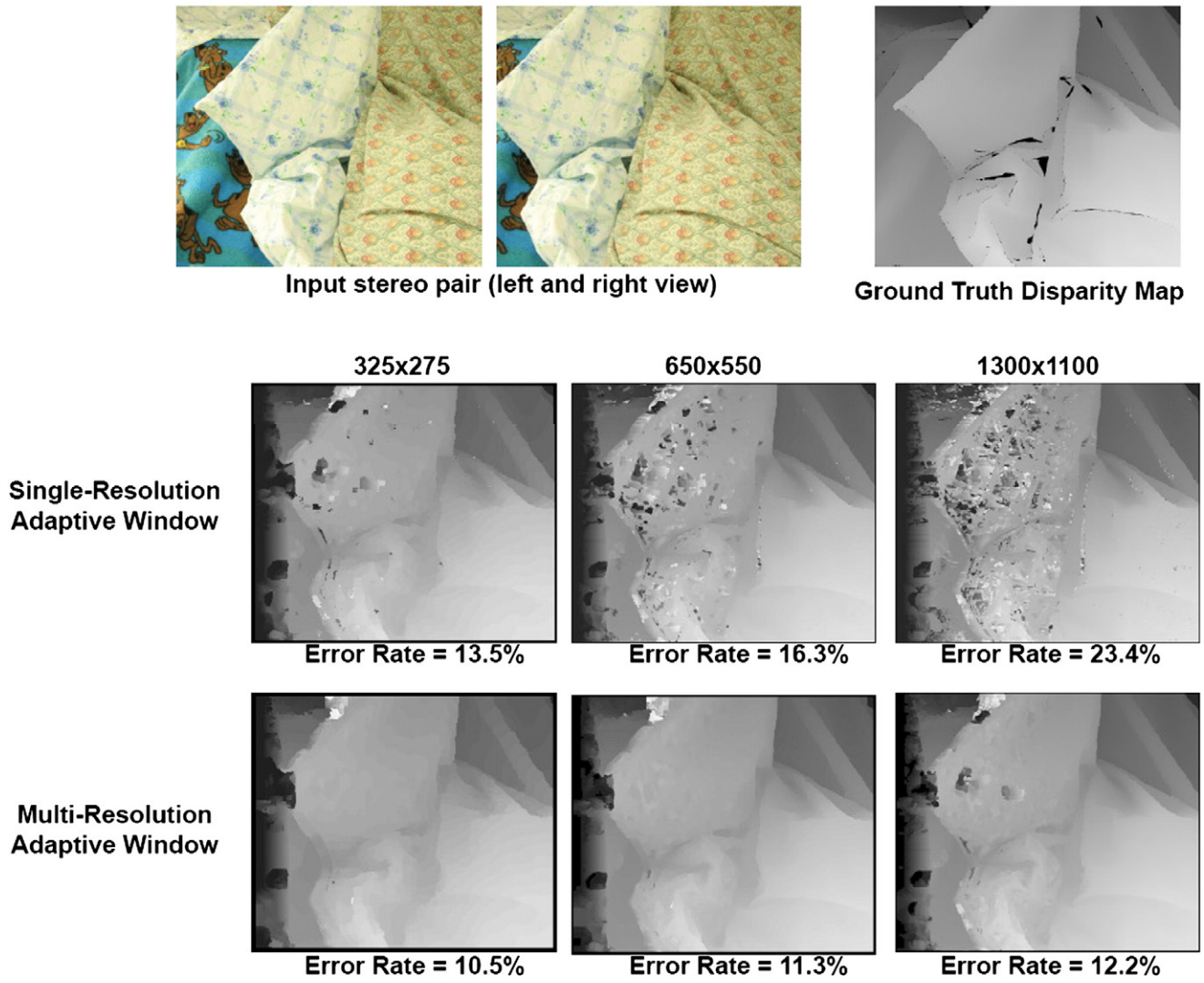


**Fig. 13.** Accuracy comparison of multi-resolution adaptive window approach and single resolution adaptive window approach. First row: stereo pairs and ground truth disparity map; second row: disparity maps obtained by the original single resolution adaptive window approach[13], on the same stereo pair at different resolutions: left — 325×275, middle — 650×550, and right — 1300×1100; third row: disparity maps obtained by our multi-resolution adaptive window approach on different resolutions: left — 325×275, middle — 650×550, and right — 1300×1100. The disparity search range is one-fourth of the image width. The error rates are displayed at the bottom of each resulting disparity map.

matching process is for each pixel of reference view, to find its correspondence in the target view. When the reference view and the target view are exchanged, the two resulting disparity maps may not be entirely identical. This problem may be caused by occlusion, reflection, differences caused by non-Lambertian surfaces, and sampling noise. Cross checking is a process to check the consistency of disparity estimation when reference view is changed. For example, after stereo matching at two different reference view, there are two resulting disparity maps: $D(left)$ and $D(right)$. For a pixel of left-view disparity map at position $(x,y)$, the disparity value is $D(left,x,y)$; this means the corresponding pixel at the right image is at position $(x+D(left,x,y),y)$; then the actual disparity value of this pixel at the right-view disparity map, which is $D(right,x+D(left,x,y),y)$, is checked; make sure that the sum of two disparity value is zero, which means the stereo matching results on this pair of pixels when disparity is estimated from two different reference views, are consistent. By doing cross checking, most of the matching errors can be effectively detected and removed, and therefore the matching accuracy can be improved. Fig. 10 shows how cross-checking removes the matching errors.

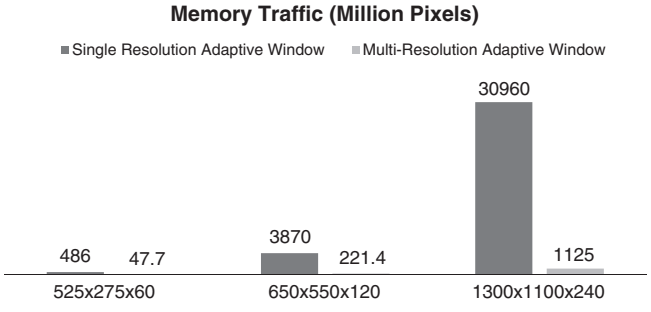


**Fig. 14.** Memory traffic comparison of multi-resolution adaptive window approach and single resolution adaptive window approach on different resolutions.

In general, cross checking can detect erroneous matching results but also leave some holes whose disparity values are missing. In a multi-resolution framework like ours, pixels from these holes at the lower resolution disparity map, will be searched at the maximum disparity range defined by the whole CTA. In practice, this would give opportunities to these holes to be filled at higher resolution.

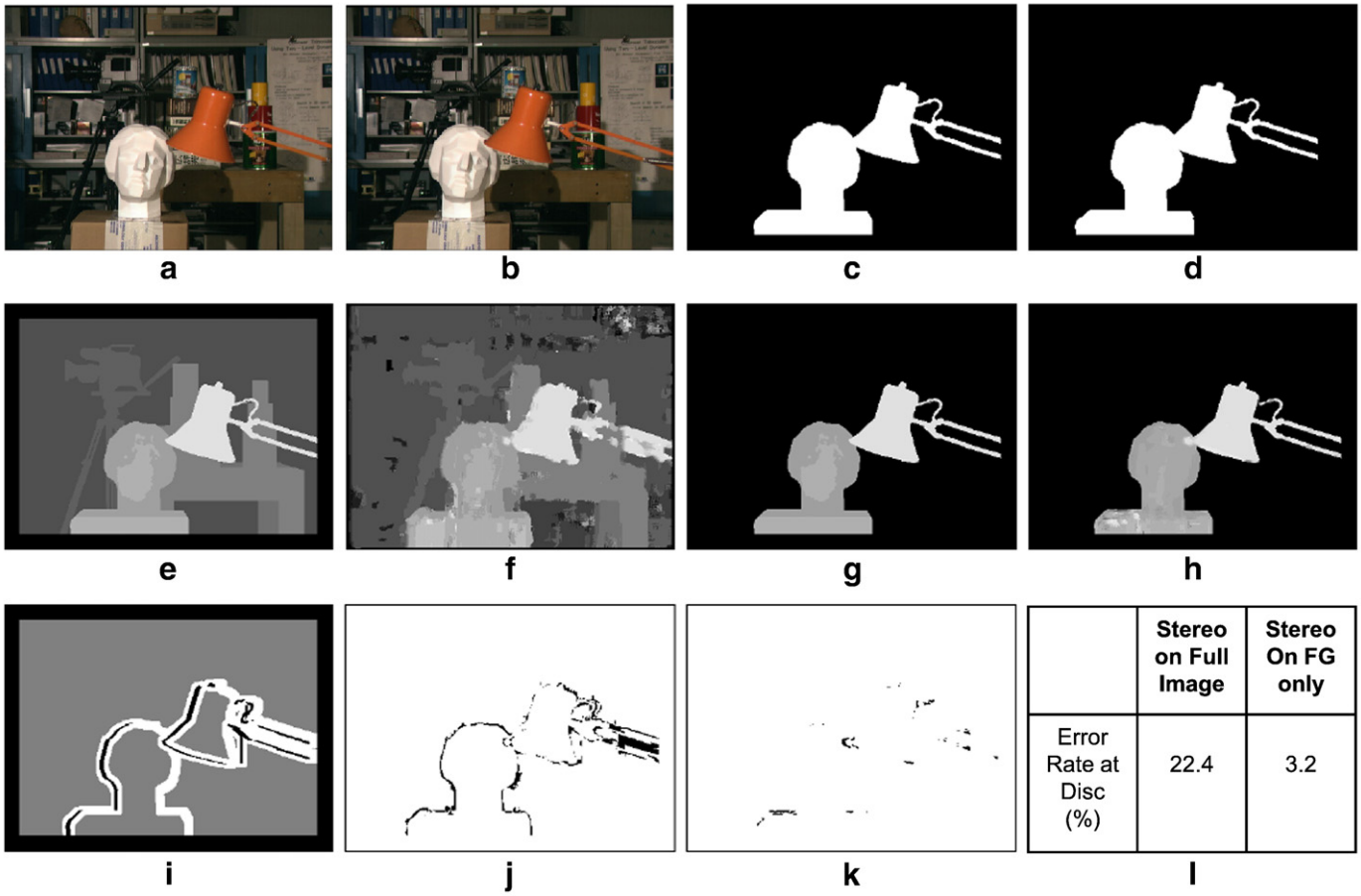


**Fig. 15.** Stereo matching on foreground region vs stereo matching on full image. (a,b) left and right view of stereo pair; (c,d) left and right view of foreground region; (e) full image ground truth disparity map; (f) full image disparity map obtained from our method without using foreground information; (g) foreground only ground truth disparity map; (h) foreground only disparity map obtained from our method using foreground information; (i) depth discontinuity map: black means boundary and occluded regions, white means boundary regions on the depth discontinuity, gray means non-occluded regions; (j,k) error map of full image stereo matching and foreground only stereo matching on the depth discontinuity regions; (l) error rate (percentage of bad pixels) only on the depth discontinuity regions around foreground blobs.

Therefore, the holes from lower resolution will not be passed or even enlarged at higher resolution.

## 4. Results

### 4.1. Middlebury evaluation

Comparing our method with other stereo algorithms is not straight forward. The reason is that our method only works on moving objects which are detected by a background model. In order to appropriately measure the performance of our method, a reference stereo video dataset with ground truth information is needed. Unfortunately, such dataset is not available. In order to compare with other regular stereo algorithms which work on a pair of static stereo images, we use the Middlebury static stereo datasets [11], with the assumption that the whole image area is foreground. The output disparity maps are shown in Fig. 11, and the error rate for different pictures are shown in Table 1.

In Table 2, the performance of 23 different stereo algorithms including our method is compared. This comparison covers all kinds of algorithms: belief propagation based algorithms, segmentation based algorithms, dynamic programming based algorithms, correlation based local algorithms, as well as some semi-global algorithms. Some of these algorithms also operate on multiple resolutions. The comparison is made both on accuracy and speed. Fig. 12 visualizes the result of this comparison on a 2-d chart. Notice that the vertical axis of this chart is in logarithmic scale. Our algorithm is at the top of this chart and is more than 100,000 times faster than the slowest algorithm. In general, it is obvious that more accurate algorithms tend to be slower. Compared to all the global or semi-global algorithms, our algorithm is much faster but also less accurate; but compared to some other local algorithms such as MultiResSSD [12] and AdaptiveWin [13], our algorithm is much faster and more accurate. This proves that the coarse-to-fine process in our multi-resolution framework works better than just averaging the matching results from multiple resolutions in [12]; and it also demonstrates that using adaptive window on multi-resolution can achieve better accuracy than AdaptiveWin [13]. On the right side of Fig. 12, we also indicate the frame-rate of stereo algorithm when applied on a $1024\times768$ stereo pair with a 256 pixel disparity range, which is used in our human detection, tracking and identification system (presented in a separate paper). Almost every algorithm faster than 0.5 fps is implemented on GPU. And our algorithm is the only one which achieves 30+ fps — a commonly used video frame-rate.

### 4.2. Multi-resolution vs single-resolution

Just like the original adaptive window approach [13], our approach is also a local algorithm, in the sense that the disparity decision for each pixel is made purely based on the local matching cost for this pixel alone. However, our method differs from the original one in the sense that the decision at each resolution is directly constrained by the decision from previous lower resolution. At the original resolution, the decision on each pixel is affected by the visual content of its neighborhood at all the different lower resolutions. Therefore,

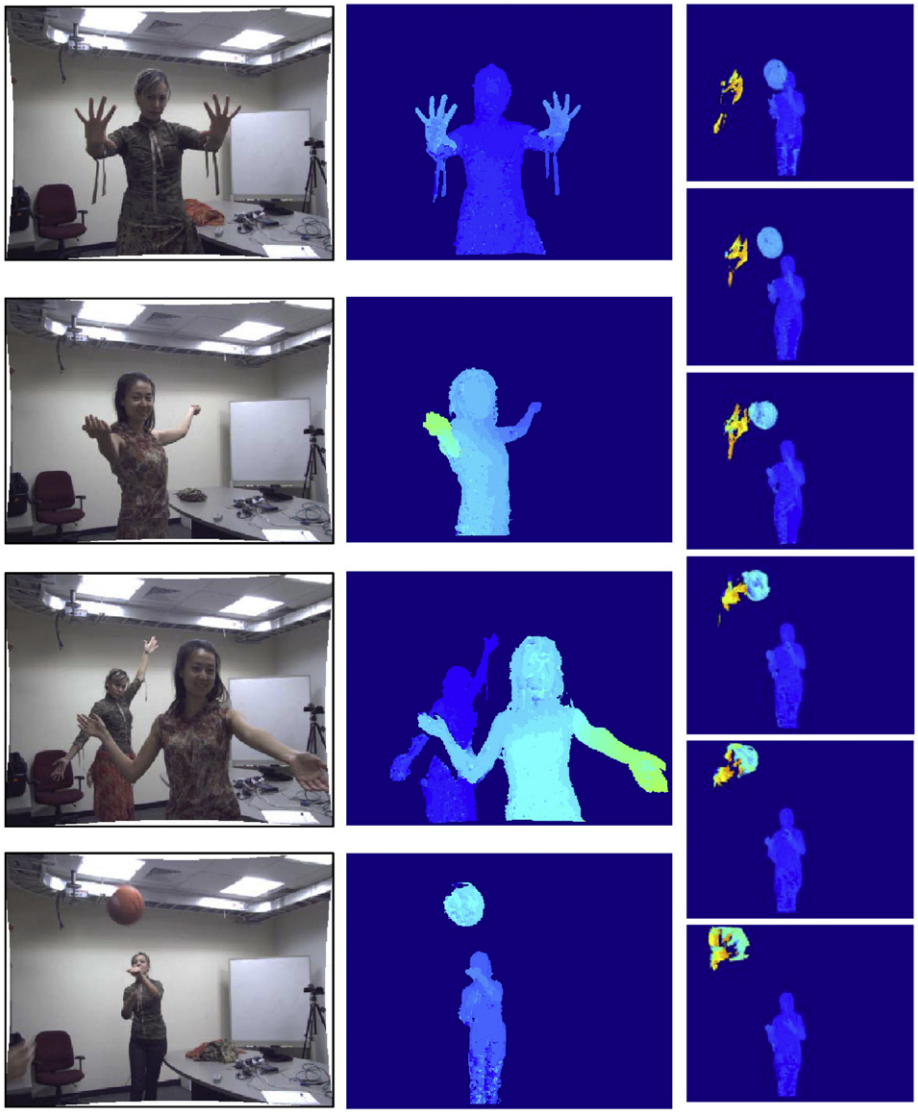

**Fig. 16.** Screen shots of our real-time stereo system working on the field.

compared to the original single resolution adaptive window approach, our approach is somewhat more "global".

This advantage is confirmed by the experimental results. In this experiment, our approach and the original approach are both applied on the same stereo image pair at three different resolutions: $325\times275$, $650\times550$ and $1300\times1100$, shown in Fig. 13. The disparity search range is always one-fourth of the image width, the same $3\times3$ pixel sub-window in both algorithms. The error rates are shown under each resulting disparity map in Fig. 13. It turns out that, at lower resolution, two algorithms produce similarly accurate result. This is due to the fact that the scene texture at that resolution is just about enough to differentiate each pixel from other pixels on the same epipolar line. As the resolution goes up, the accuracy of the original algorithm deteriorates dramatically, while the accuracy of our algorithm only slightly drops. Increasing the size of the sub-window does not help the original algorithm because it brings too much error at the depth discontinuity region of the image.

Improved accuracy is not the only advantage of our approach over the original one. Our approach is also much faster than the original approach does. The reason is obvious: in our approach, a full range disparity search is applied only at the lowest resolution. On higher resolutions, disparity search only occurs in a 4 pixel range for each pixel. This significantly reduces the computational cost. As we know, on a SIMD architecture GPGPU device, memory traffic is the most significant part of computing cost for most processing tasks. The comparison of memory traffic on our approach and the original approach is shown in Fig. 14. Again the experiment is done on three different resolutions, the disparity search range is one-fourth of the image width. At the original resolution, which is $1300\times1100$ pixel, our approach is about 30 times cheaper than the original approach on memory access.

Therefore, the effectiveness of using multi-resolution on a purely stereo algorithm is clearly proven by the significant improvement on both accuracy and speed.

### *4.3. Stereo on foreground vs stereo on full image*

The reason that we use foreground detection to constrain our stereo matching algorithm is that only moving objects are of interest in our target applications. Therefore we don't have to waste computing resource on those uninteresting background regions of image. In reality, indoor background is usually poor of texture and therefore is a major source of the overall matching errors. Many previous applications use a full image stereo algorithm then use the foreground detection result to filter the full image depth-map. However, we think using foreground detection result at the early

stage of stereo matching algorithm would be more efficient both in cost and quality.

Computing stereo matching only on foreground region not only saves computing time, it also improves the accuracy of the stereo matching, especially at the boundary regions of foreground blobs. These regions are also the depth discontinuity regions between foreground objects and their background. The reason is simple: without foreground information, we have to make a decision at the depth discontinuity region: which side each pixel belongs to? This decision is often difficult for the local algorithms because the matching window they use is usually bigger than one pixel. Foreground information can make this decision very easy: all foreground pixels belong to the foreground region.

In order to verify this advantage, the following experiment is conducted: some foreground objects (the lamp and the statue) in one of the Middlebury benchmark stereo images — Tsukuba, as shown in Fig. 15(c) and (d), are manually segmented out. The depth discontinuity region (Fig. 15(i)) around the foreground objects is also manually defined. First our multi-resolution adaptive window approach is applied on the full image area, to obtain the full image disparity map (Fig. 15(f)); then our stereo matching algorithm with the constraint that only foreground pixels can be matched to each other is applied, and the foreground disparity map shown in Fig. 15(h) is obtained. Finally the error rates of two different results only at the discontinuity region are compared. The error maps are shown in Fig. 15(j) and (k) and the error rates are listed in Fig. 15(l). This experiment clearly shows that using foreground information significantly improves the quality of disparity map on foreground regions that we care. The disparity values on the boundary of these objects are smooth and accurate. Some screen-shots of our system working with live video in real-time is shown in Fig. 16.

## 5. Conclusions

We have presented a new stereo matching algorithm and GPGPU-based implementation, which is focused on real-time applications such as motion detection, object tracking, interactive user interface, pose analysis, etc. We carefully implemented it on commodity computer hardware using CUDA. Our implementation can deal with high definition video, large disparity searching range, and speeds significantly higher than the input video stream frame rates. Therefore, a lot of GPU and CPU processing resource can be allocated to other processing tasks. This technique will enable the use of high quality stereo matching in many applications.

In the past, stereo video cameras were often expensive and hard-to-use, compared to the traditional single sensor cameras. The main reason has being that compute-intensive real-time processing of stereo data was too expensive for the existing hardware. Because of the rapid advance of hardware performance, and a careful algorithm design, this situation has changed. We believe that stereo cameras should be able to serve most computer vision applications as a kind of ubiquitous visual sensors, and regard our work as a contribution towards that goal.

There are some shortcomings of our algorithm. Our algorithm uses background modeling, which makes it unable to work with scenarios where background model is not applicable, such as moving cameras. Because it is still a local algorithm, its accuracy performance on poorly textured objects is not good. Therefore it is not an ideal solution for applications where accuracy is the major concern. But for applications where speed performance is the key concern, such as real-time surveillance application, our algorithm works very well.

## References


[1] N. Cornelis, L.V. Cool, Fast scale invariant feature detection and matching on programmable graphics hardware, Proceedings of Conference on Computer Vision and Pattern Recognition Workshop on CVGPU, 2008.

[2] Y. Luo, R. Duraiswami, Canny edge detection on NVIDIA CUDA, Proceedings of Conference on Computer Vision and Pattern Recognition Workshop on CVGPU, 2008.

[3] M. Gong, Y.-H. Yang, Near real-time reliable stereo matching using programmable graphics hardware, Proceedings of Conference on Computer Vision and Pattern Recognition, 2005, pp. 924–931.

[4] M. Gong, R. Yang, Image-gradient-guided real-time stereo on graphics hardware, Proceedings of the Fifth International Conference on 3-D Digital Imaging and Modeling, 2005, pp. 548–555.

[5] N. Cornelis, L.V. Cool, Real-time connectivity constrained depth map computation using programmable graphics hardware, Proceedings of Conference on Computer Vision and Pattern Recognition, 2005, pp. 1099–1104.

[6] S.J. Kim, D. Gallup, J.M. Frahm, A. Akbarzadeh, Q. Yang, R. Yang, D. Nister, M. Pollefeys, Gain adaptive real-time stereo streaming, Proceedings of International Conference on Computer Vision System, 2007.

[7] J. Gibson, O. Marques, Stereo depth with a unified architecture GPU, Proceedings of Conference on Computer Vision and Pattern Recognition Workshop on CVGPU, 2008.

[8] D. Gallup, J.M. Frahm, P. Mordohai, Q. Yang, M. Pollefeys, Real-time plane-sweeping stereo with multiple sweeping directions, Proceedings of Conference on Computer Vision and Pattern Recognition, 2007.

[9] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, D. Nister, Real-time global stereo matching using hierarchical belief propagation, BMVC '06, 2006, p. III:989.

[10] NVIDIA CUDA programming guide 2.0.

[11] D. Scharstein, R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, International Journal of Computer Vision, 47(1), 2002, pp. 7–42.

[12] R. Yang, M. Pollefeys, Multi-resolution real-time stereo on commodity graphics hardware, Proceedings of Conference on Computer Vision and Pattern Recognition, 2003, pp. 211–218.

[13] R. Yang, M. Pollefeys, Improved real-time stereo on commodity graphics hardware, Proceedings of Conference on Computer Vision and Pattern Recognition Workshop on Real-time 3D Sensors and Their Use, 2004.

[14] U. Park, A. Jain, I. Kitahara, K. Kogure, N. Hagita, Vise: visual search engine using multiple networked cameras, Proceedings of IEEE Computer Society Conference on Pattern Recognition, 2006, pp. 1204–1207.

[15] C. Stauffer, W. Grimson, Adaptive background mixture models for real-time tracking, IEEE Conference on Computer Vision and Pattern Recognition, 1999, pp. 246–252.

[16] S.D. Cochran, G. Medioni, 3D surface description from binocular stereo, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1992, pp. 981–994.

[17] Y. Taguchi, B. Wilburn, C.L. Zitnick, Stereo reconstruction with mixed pixels using adaptive over-segmentation, IEEE Conference on Computer Vision and Pattern Recognition, 2008.

[18] J. Sun, Y. Li, S. Bing, K.H. yeung Shum, Symmetric stereo matching for occlusion handling, IEEE Conference on Computer Vision and Pattern Recognition, 2005, pp. 399–406.

[19] Q. Yáng, L. Wang, R. Yang, H. Stewénius, D. Nister, Stereo matching with color-weighted correlation, hierarchical belief, CVPR, IEEE Computer Society, 2006, pp. 2347–2354.

[20] E. Larsen, P. Mordohai, M. Pollefeys, H. Fuchs, Temporally consistent reconstruction from multiple video streams using enhanced belief propagation, In Proceedings of IEEE International Conference on Computer Vision, 2007, pp. 1–8.

[21] S. Mattoccia, A locally global approach to stereo correspondence, In Proceedings of 3DIM, 2009.

[22] Z. Wang, Z. Zheng, A region based stereo matching algorithm using cooperative optimization, IEEE Conference on Computer Vision and Pattern Recognition, 2008, pp. 1–8.

[23] A. Klaus, M. Sormann, K. Karner, Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure, ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition, IEEE Computer Society, Washington, DC, USA, 2006, pp. 15–18.

[24] H. Hirschmuller, Stereo vision in structured environments by consistent semi-global matching, Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 2, 2006, pp. 2386–2393.

[25] F. Tombari, S. Mattoccia, E. Addimanda, Near real-time stereo based on effective cost aggregation, ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition, 2006.

[26] J. Salmen, M. Schlipsing, J. Edelbrunner, S. Hegemann, S. Luke, Real-time stereo vision: making more out of dynamic programming, CAIP '09, 2009, pp. 1096–1103.

[27] Y. Deng, X. Lin, A fast line segment based dense stereo algorithm using tree dynamic programming, ECCV '06, 2006, pp. 201–212.

[28] S. Kosov, T. Thormahlen, H. Seidel, Accurate real-time disparity estimation with variational methods, ISVC '09, 2009, pp. 796–807.

[29] L. Wang, M. Liao, M. Gong, R. Yang, D. Nister, High-quality real-time stereo using adaptive cost aggregation and dynamic programming, 3DPVT '06: Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 798–805.

[30] Q. Yang, C. Engels, A. Akbarzadeh, Near real-time stereo for weakly-textured scenes, BMVC '08, 2008.

[31] S. Forstmann, Y. Kanou, J. Ohya, S. Thuering, A. Schmitt, Real-time stereo by using dynamic programming, CVPRW '04: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04), Volume 3, IEEE Computer Society, Washington, DC, USA, 2004, p. 29.

[32] W. Yu, T. Chen, F. Franchetti, J. Hoe, High performance stereo vision designed for massively data parallel platforms, IEEE Transactions on Circuits and Systems for Video Technology, 2010.
[33] N. Cornelis, L.V. Gool, Real-time connectivity constrained depth map computation using programmable graphics hardware, CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), Volume 1, IEEE Computer Society, Washington, DC, USA, 2005, pp. 1099–1104.
[34] Q. Yang, L. Wang, N. Ahuja, A constant-space belief propagation algorithm for stereo matching, IEEE Conference on Computer Vision and Pattern Recognition, 2010.
[35] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, M. Vincze, A fast stereo matching algorithm suitable for embedded real-time systems, Computer Vision and Image Understanding, 2010.