Stereo Depth with a Unified Architecture GPU

Joel Gibson and Oge Marques
Department of Computer Science and Engineering
Florida Atlantic University, Boca Raton, FL

Abstract

This paper describes how the calculation of depth from stereo images was accelerated using a GPU. The Compute Unified Device Architecture (CUDA) from NVIDIA was employed in novel ways to compute depth using BT cost matching and the Semi-Global Matching algorithm. The challenges of mapping a sequential algorithm to a massively parallel thread environment and performance optimization techniques are considered.

1. Introduction

The calculation of a depth field from stereo images is a topic of active research (2). Depth information is generally derived by trying to match pixels in one of the stereo images with the corresponding feature in the other image. Generally a cost is computed based on some measure of similarity. This result alone is not generally usable but must be further refined. Many strategies have been developed, with the best results usually requiring a large amount of processing time.

There are applications which require real time or near real time depth feedback. Examples include machine vision in industrial processes where feedback for mechanical control is required. Real time depth feedback is desired in robotics for navigational decision making.

There has also been a recent resurgence in stereo theatrical releases. This requires post production tools that work interactively with their operator. In these types of applications, speed is more important than precision.

In general, computing a depth field is a computationally challenging task. Computing depth on a CPU has generally been a slow process. The quicker efforts produce inaccurate results while quality depth results have been attainable but have been extremely slow. Efforts to accelerate these calculations have focused on multimedia type optimizations using the SIMD module in consumer CPUs. Recently, the Graphics Processing Unit (GPU) (13) has received much attention. Driven by a widespread gaming market, the performance of GPUs has grown much faster than Moore's law. According to Owens, et al (14) graphics hardware performance has doubled every six months. Until recently, efforts to channel the brute horsepower of the GPU has been limited to problems that could be made to look like a 3D description to be rendered by the GPU. The resulting "image" contained the solution(s) that were sought. In the last year GPU manufacturer Nvidia (3) has made a concerted effort to produce a more general purpose programming environment (4). This creates opportunity for a parallel task like depth computation to be performed faster. Also, these new GPUs natively support floating point processing, offering the chance for higher quality results.

The goal of this research is to a give an overview of stereo algorithms, identify the best candidate for GPU acceleration, and lastly implement the selected algorithm on a GPU.

2. Background and Related Work

Before delving into the algorithm and implementation chosen, first we will cover some background on depth algorithms and contemporary GPU architecture.

2.1. Depth Algorithms

The definitive survey and taxonomy of stereo depth algorithms was presented by Scharstein and Szelinski in 2002 (16). While the appearance of new algorithms is leaving the paper somewhat dated, it still remains the most comprehensive survey paper. Their taxonomy identifies the following steps:

1. Rectification
2. Matching Cost Calculation
3. Cost Aggregation
4. Disparity Computation/Optimization
5. Disparity Refinement

978-1-4244-2340-8/08/$25.00 ©2008 IEEE

Rectification is generally required if the epipolar lines[1] do not match the scan lines. This can be performed as a preprocessing step and is not covered as part of this development. Most depth algorithms require rectified images as a prerequisite. It should be noted that if this correction is required, a GPU is a natural choice since it was natively designed to do image transforms.

The Matching Cost Calculation can be something as direct as the Sum of Absolute Differences (SAD) or Sum of Square Differences (SSD). The cost calculation chosen in this project is really a sampling insensitive enhancement of the SAD metric. Hierarchal Mutual Information (HMI) (10) is a more sophisticated metric which produces superior results to SAD and SSD.

The Aggregation stage typically involves convolving a window or averaging costs over an area. This is generally a low complexity attempt to smooth the Matching Cost Calculation. Not all algorithms employ this step.

Disparity Computation and Optimization is broadly divided into global or local algorithms. The simplest local algorithm is winner-take-all. Global optimizations generally try to minimize a global energy metric. Examples of global methods include Graph Cuts (9) and Belief Propagation (20). The global methods generally produces the best results however they may take 2-3 orders of magnitude more processing time than a local method.

Refinement involves cleaning up after the optimization step. This typically involves independently generating disparities[2] from the left image to the right image and separately from the right image to the left image. These results are then cross checked for consistency. Disparities that do not match are typically invalidated. This produces holes in the disparity map which may be filled by interpolation. This disparity map is inversely proportional to the depth map of the image.

2.2. Graphics Processing Units

In the modern graphics pipeline, the Vertex Processor and Fragment Processor are programmable elements. These GPUs suffer from the limitation that while reads can be random, writes are always in raster order. There was also no way for the parallel threads to communicate with each other during the execution of their code. Furthermore, the program size was very limited and instruction set was missing ordinary integer bitwise operators.

In 2007 NVIDIA introduced the Compute Unified Device Architecture (CUDA). It was called unified because it combined the Vertex Processor and the Fragment Processor into a single general purpose programmable element. The generic CUDA model is shown in Figure 1. The first production model was called the G80 and consisted of sixteen multiprocessors. Each multiprocessor has eight processing elements. The multiprocessor has only a single instruction execution unit for all eight processing elements. This is not strictly a SIMD processor because the thread executing on each processing element within a multiprocessor can diverge, albeit at reduced efficiency. Each thread has a unique thread ID which consists of an (x, y, z) value. This thread ID is what allows each thread to process different data or behave differently than the other threads. A key advantage that the G80 GPU (1) brings is that reads and writes are random. In addition, there is 16kB of shared memory per multiprocessor that is accessible from any of the threads within a given block. This is a key feature to allow threads to synchronize and share information between each other.

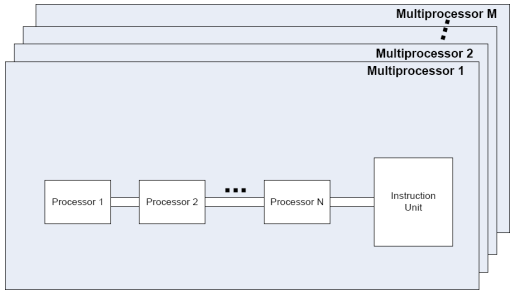

Figure 1: CUDA multiprocessor.

Each GPU program run from the host computer is called a kernel. A kernel is run on a grid. The size and geometry of the blocks with a grid and the threads within a block are configured by the programmer at run time (5).

In order to map any algorithm effectively to a GPU, there must exist a reasonably efficient parallel expression of that algorithm. There must also be a nontrivial amount of computation relative to each main memory access. Device memory is not cached in the GPU. The processor instead executes hundreds of threads concurrently. It relies on efficient hardware thread switching to execute code in active threads while waiting for memory requests to be serviced on behalf of threads that have been put to sleep. The execution of the these hundreds or thousands of threads concurrently hides the memory access latency as long as there is sufficient computation to be done.

2.3. Related Work

Efforts to use GPUs for depth calculations are not new. Yang, et al (22) used a GPU in a multi-view environment to compute SADs, perform an area agregation, and followed by a winner-take-all. A year later, Yang and Pollefeys (21)

[1]The plane that includes the object in 3D space, and the corresponding points on the left and right image form the epipolar plane. The intersection of the epipolar plane and the left and right image planes form the epipolar lines (18).

[2]Disparity is defined as the distance in position between the same object in the left image and right image.

Figure 2: Left and Right cones test image.

used an SSD cost with a multi-resolution approach to obtain near real time performance for camera images. Woetzel and Koch (19) applied SSDs using a GPU and then used the texture mapping capability to perform bilinear interpolation while searching for sharp edges to preserve.

The most relevant previous effort was by Rosenberg, Davidson, et al (15) who adapted Hirschmüller's Semi-Global cost method to a previous generation GPU. They achieved near real time with 160×120 camera images. However, the number of sweeps was limited and accuracy seems constrained to 8 bit calculations. Their results were encouraging that this type of algorithm could be mapped to a GPU environment.

3. Depth Implementation on a GPU

In choosing a depth algorithm to map to a GPU, the constraints of the previous section were carefully considered. First, the computation must be capable of being represented as massively parallel. Second, accesses to device memory needs to be minimal. Third, the best quality possible is desired while still running at or near real-time speed.

Of the twenty or so best algorithms listed on the Middlebury Stereo site (2), almost all are of the Global category. These algorithms include Belief Propagation, Graph Cuts, and Segmentation. These seemed to lack any obvious parallel implementation. An exception which stood out was Hirschmüller's Semi-Global Matching(SGM) (10). The SGM competes favorably with the global cost matching algorithms but it has a cost complexity $O(Width \times Height \times dispMax)$, on the same order as local methods. Local methods, while inherently parallel do not, on their own, produce results of sufficient quality to be considered.

The SGM algorithm performs cost optimization. It does not depend on a particular cost matching function first. Hirschmüller (10; 11; 12) showed results with the Birchfield and Tomasi (BT) (6) as well as a Hierarchal Mutual Information (HMI) cost calculation. While the HMI produces somewhat better results, the BT algorithm was selected based on its lower implementation complexity. The bulk of the effort then became how to implement and optimize the SGM function.

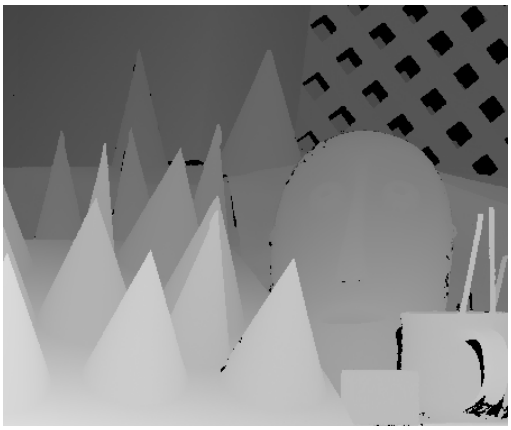

Figure 3: Ground truth depth of Left Cone image in Figure 2.

Since this generation of GPU is natively optimized to work in floating point arithmetic, the image is input as planar field of RGB floats. This simplifies programming, speeds development, and generally improves calculation accuracy compared to integer math.

Figure 2 shows the reference pair of stereo images from the Middlebury Stereo Data Set (17) that were used in these trials. Figure 3 shows the ground truth of the left cone image.

3.1. BT Cost Calculation

Direct SAD cost calculation suffers from sampling sensitivity. This means that an image feature present on a pixel in the left image does not necessarily fall squarely on a pixel in the right image, and vice versa. The BT cost calculation removes the sampling sensitivity from the cost function while adding only an incremental increase to the calculation.

Birchfield and Tomasi define the value of a sub-sampled pixel between the current pixel and it's neighbor to the left as I_R^-. Similarly, I_R^+ represents a sub-sampled pixel to the right of the reference pixel.

$$I_R^- = \frac{1}{2}(I_R(x_R) + I_R(x_R - 1)) \quad (1)$$

$$I_R^+ = \frac{1}{2}(I_R(x_R) + I_R(x_R + 1)) \quad (2)$$

Where I_R^- represents the value of a sub-sampled pixel to the left of the reference pixel and I_R^+ is a sub-sampled pixel to the right of the reference pixel.

A minimum and a maximum of the reference pixel intensity and its neighbors are calculated for each reference pixel.

$$I_{min} = \min\{I_R^-, I_R^+, I_R(x_R)\} \quad (3)$$

$$I_{max} = \max\{I_R^-, I_R^+, I_R(x_R)\} \quad (4)$$

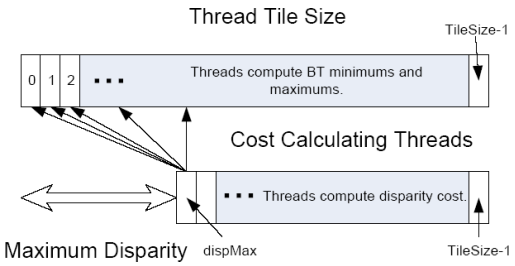

Figure 4: Thread allocation for BT cost calculation.

The cost function is then calculated according to:

$$C(\mathbf{p}, d) = \max\{0, I_L(x_L) - I_{max}, I_{min} - I_L(x_L)\} \quad (5)$$

where $C(\mathbf{p}, d)$ represents the cost or degree of dissimilarity between the current pixel x_L and a reference pixel that is some disparity d pixels away. A symmetrical calculation from the right image to left is required and the minimum of the two disparities is taken as the final cost.

As can be seen, each maximum or minimum of the reference image is accessed $dispMax$ times, where $dispMax$ the maximum disparities computed. If these intermediate values were stored in main device memory, as would have been the case with past GPU architectures, every disparity calculation would represent another trip to main memory creating the bottleneck for a fairly simple calculation. Here we create a tile of threads $TileSize$ wide. The upper bound of the $TileSize$ is limited by the amount of shared memory available. This implementation required seven buffers of $TileSize$ width in shared memory space. First a tile width's of a reference line component (R,G, or B) is read in to shared memory, then the maximums and minimums are computed as shown in Equations 3 and 4. Each RGB component has a separate maximum and minimum calculation and storage, hence three maximums, three minimums, plus an additional temporary buffer for the results of Equations 1 and 2 which are combined into a single computational step.

After each thread in the tile computes a maximum and minimum for each color component, a `syncthreads()` call is made. This is a barrier synchronization that ensures all of the threads have computed the task before any can move on. This is vital here because there are in general far more virtual threads than there is physical hardware. Therefore, it is possible some threads have finished their task before other threads have started.

Coming out of the `syncthreads()`, the cost computation can begin. The cost of each color component is computed separately and summed together for an aggregate cost. To compute the cost, all the threads can not be used. Since each thread has a current pixel that requires a reference pixel up to $dispMax$ positions away, the $dispMax$ threads on the left of the tile, for the case of computing the left image disparity, those $dispMax$ threads cannot reach a reference pixel. Hence, each tile block of threads can only compute $TileSize - dispMax$ number of costs as shown in Figure 4. This sets an upper limit on the efficiency of this calculation of $(TileSize - dispMax)/TileSize$.

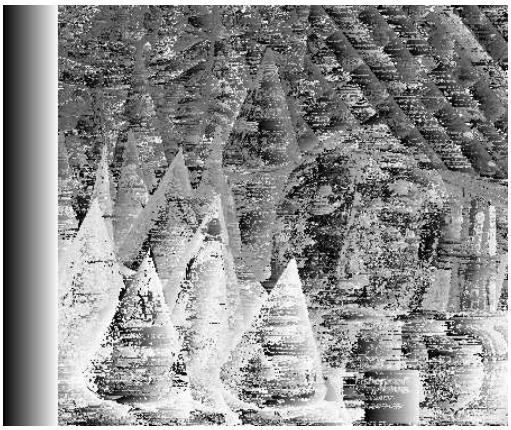

Figure 5: Output of Birchfield and Tomasi Cost Calculation with winner-take-all.

This calculation creates a three dimensional cost array that is $Width \times Height \times dispMax$. Because of its size this has to be created in device memory. Each (x, y) image location in this cost table has an array of dispMax elements associated with it. The k^{th} element has the cost of a disparity k. A shortcut to a disparity result is to find the array location with the least cost. The array location becomes the disparity. This is called a winner-take-all (WTA) approach and produces a noisy but recognizable result in Figure 5.

3.2. SGM Cost Optimization

Semi-Global cost matching attempts to minimize the following global energy equation.

$$E(D) = \sum_{\mathbf{p}} C(\mathbf{p}, D_{\mathbf{p}}) + \sum_{q \in N_{\mathbf{p}}} P_1 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| = 1] + \sum_{q \in N_{\mathbf{p}}} P_2 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| > 1] \quad (6)$$

where D represents the disparity image. $D_{\mathbf{p}}$ corresponds to all of the disparities computed for a particular pixel and $D_{\mathbf{q}}$ represents the disparities of pixels within a neighborhood q of pixel p. The first term sums the costs of all disparities for the given pixel p. These costs summed are the results of the BT cost calculation. The second term gives a penalty P_1 to neighboring pixels which have a disparity that differs from the current pixel by only one. The last term adds a

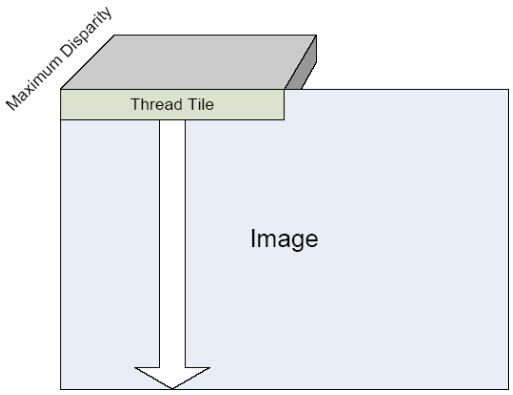

Figure 6: Thread allocation for computing L_r, top to bottom pass.

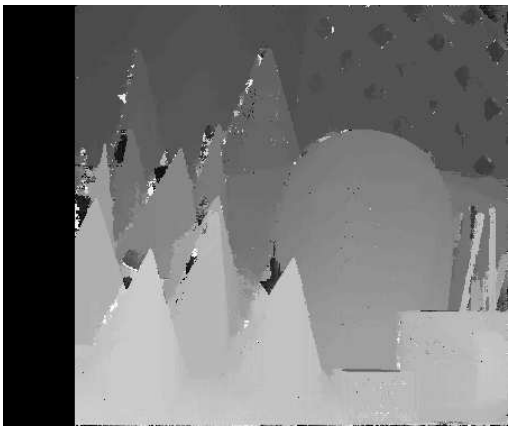

Figure 7: Result of Semi-Global cost optimization.

larger penalty P_2 if the neighbors disparity is different by more than one from the current pixel's disparity. However, to avoid smearing the edges of objects, this penalty is actually calculated as $P_2 = \frac{P_2'}{I_{\mathbf{p}} - I_{(\mathbf{p}-\mathbf{r})}}$, where the penalty P_2' is mitigated by the difference in intensities of the current pixel and the neighbor in question. This assumes that object's edges usually have a change in intensity when accompanied by a sharp disparity change.

This global energy minimization is realized by recursively computing Equation 7 across r paths where r varies over $[1, 8]$ in our implementation.

$$\begin{aligned} L_{\mathbf{r}}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min\{ & L_r(\mathbf{p} - \mathbf{r}, d), \\ & L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d-1) + P_1, \\ & L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d+1) + P_1, \\ & \min_i L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, i) + P_2\} - \\ & \min_k L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, k) \end{aligned} \tag{7}$$

$L_{\mathbf{r}}(\mathbf{p}, d)$ represents the cost of disparity d calculated at the current pixel along path $\mathbf{r}$. The final term $\min_k L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, k)$ is subtracted out so that the cost does not continually grow as $L_{\mathbf{r}}(\mathbf{p}, d)$ is calculated along the path.

The $L_{\mathbf{r}}$ values from all of the paths are summed to produce a smoothed, optimized cost table $S(\mathbf{p}, d)$.

$$S(\mathbf{p}, d) = \sum_{\mathbf{r}} L_{\mathbf{r}}(\mathbf{p}, d) \tag{8}$$

The results are shown in Figure 7.

4. Experimental Results

Each path in Equation 7 starts on an image border pixel. The cost $S(\mathbf{p}, d)$ at the border is set to the value of the computed cost $C(\mathbf{p}, d)$. The path is then traversed away from the border with each pixel's disparity costs computed based on the previous pixel's costs.

While this is intrinsically a serial computation there are opportunities for parallelization which we utilize. First, the threads are tiled in the z axis. These are used to simultaneously compute all of the disparities of a single pixel in parallel. Secondly, the thread blocks are tiled in either a column or row depending on the direction of the path. For example, for the sweep from top to bottom of the image, the thread blocks are tiled across the top row of the image and all swept top to bottom together. This is shown in Figure 6.

However, there are challenges with this strategy. The $\min_i L_r(\mathbf{p} - r, i)$ value in Equation 7 is the minimum cost of all the previous pixel's disparities. Since all of the disparities are calculated by separate threads no one knows the minimum. Since all of the previous pixel's disparities are used by the current calculation, saving all the previous pixel's disparities in shared memory is a good strategy. With the previous disparity cost array in shared memory, a parallel sort routine can be used to find the minimum. Since there are relatively few values in the disparity array initially the Bitonic Sort algorithm (8) was used. Although this is a very efficient sorting algorithm, at a disparity depth of 64, this sort was consuming about 300 ms of time in the aggregate. This represents sorting 64 values for each pixel a total of 8 times. Replacing this sort with even a naïve scan tree (7) searching for the minimum saved 200 ms as shown in Table 1.

The calculation of the P_2 cost which is based on the inverse of the change of intensity from the previous pixel to the current was very inefficient in the initial implementation. Since each thread is independently calculating it's own disparity cost, each thread would go to device memory to get the pixel intensity values. This initially created a 100 ms overhead that was eliminated by reading the pixel values once per pixel, and computing the associated cost only with

Kernel Call	Original Time(ms)	Optimized Time(ms)
BT Cost Calculation	42.4	42.4
Semi-Global Calculation(8x)	455.1	122.7
Find Min Disparity	300.0	76.7
Remainder of SGM Calc	155.1	46.0
Winner-Take-All	5.7	5.7
Total	503.3	170.8

Table 1: Execution times for 450×375 RGB float images with $dispMax = 64$.

the first thread. This single float was then saved to shared memory where it is efficiently retrieved by all the threads.

5. Conclusion

Depth from stereo images while not an embarrassing parallel computation, has opportunities for parallelism. Recent GPU developments enable more general parallel computing than was possible in the past. Harnessing the GPU for depth calculation was shown in this experiment to be effective. Performing Semi-Global Cost Computation with floating point on the G80 NVIDIA GPU was demonstrated to execute at least 4× faster than optimized SIMD code on a 2.8 GHz Xeon processor (10). Other than ease of calculation, accuracy benefits of floating point versus fixed point calculations were left for future work.

There are further opportunities for optimization in the naïve scan tree which, although dramatically improved as shown in Table 1, still accounts for half of the SGM calculation time. Although not real time, this near real time speed will still yield interactively adequate performance in some applications. There is further opportunity for quality improvement by implementing Hirschmüller's HMI cost matching function instead of BT cost matching. Additional improvements include implementing a refinement stage. This would require median filtering, left-right cross checking, and filling invalid disparities. Excellent performance should be obtained performing these steps on this GPU.

References

[1] http://developer.nvidia.com/object/cuda.html. 2

[2] http://vision.middlebury.edu/stereo/eval/. 1, 3

[3] http://www.nvidia.com/. 1

[4] http://www.nvidia.com/object/cuda_home.html. 1

[5] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 1.0 edition, June 2007. 2

[6] S. Birchfield and C. Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Transactions on Pattern Analysis and Machine Intellignce*, Vol. 20, No. 4, 1998. 3

[7] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1990. 5

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998. 5

[9] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, November 2001. 2

[10] H. Hirschmüller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2005. 2, 3, 6

[11] H. Hirschmüller. Stereo vision in structured environments by consistent semi-global matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2006. 3

[12] H. Hirschmüller. Evaluation of cost functions for stereo matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2007. 3

[13] M. Macedonia. The gpu enters computings mainstream. *Computer*, 36(10):106–108, October 2003. 1

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005. 1

[15] I. D. Rosenberg, P. L. Davidson, C. M. R. Muller, and J. Y. Han. Real-time stereo vision using semi-global matching on programmable graphics hardware. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, 2006. 3

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

[17] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003. 3

[18] L. G. Shapiro and G. C. Stockman. *Computer Vision*. Prentice Hall, 2001. 2

[19] J. Woetzel and R. Koch. Real-time multi-stereo depth estimation on gpu with approximative discontinuity handling. In *1st European Conference on Visual Media Production*, March 2004. 3

[20] Q. Yang, L. Wang, R. Yang, H. Stewenius, and D. Nister. Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR) 2006*, volume 2, pages 2347– 2354, 2006. 2

[21] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003. 2

[22] R. Yang, G. Welch, and G. Bishop. Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proceedings of the 10 th Pacific Conference on Computer Graphics and Applications*, 2002. 2