# Improved Real-Time Stereo on Commodity Graphics Hardware

Ruigang Yang
Department of Computer Science
University of Kentucky
Lexington, Kentucky, USA

Marc Pollefeys
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina, USA

Sifang Li
Department of Computer Science
University of Kentucky
Lexington, Kentucky, USA

## Abstract

This paper presents a detailed description of an advanced real-time correlation-based stereo algorithm running completely on the graphics processing unit (GPU). This is important since it allows to free up the main processor for other tasks including high-level interpretation of the stereo results. Compared to previous GPU-based stereo implementations our implementation includes some advanced features such as adaptive windows and cross-checking.

By taking advantage of advanced features of recent GPUs the proposed algorithm is also a lot faster than previous implementations. Our implementation running on an ATI Radeon 9800 graphics card achieves over 289 million disparity evaluations per second including all the overhead to download images and read-back the disparity map, which is several times faster than commercially available CPU-based implementations.

## 1 Introduction

Depth from stereo has traditionally been, and continues to be one of the most actively researched topics in computer vision. While some recent algorithms have obtained excellent results by casting the stereo problem as a global optimization problem, real-time applications today have to rely on local methods, most likely correlation-based ones, to obtain dense depth maps in real time and online.

It is only recently that real-time implementations of stereo vision became possible on commodity PCs, with the help of rapid progress in CPU clock speed, and assembly level optimizations utilizing special extensions of the CPU instruction set, such as the MMX extension from Intel. While it is a tremendous achievement that some of them could perform in the order of 100 million disparity estimations per second (Mde/s) in software [9, 10, 11, 13][1], there are few CPU cycles left to perform other tasks including high-level interpretation of the stereo results. In many real-time applications, such as robot navigation, to calculate a raw depth map is only the first step in the entire processing pipeline.

Recently, driven by consumer demands for better realism in computer-generated images, the graphic processing unit (GPU) on the graphics board has become increasingly programmable, to the point that it is now capable of efficiently executing a significant number of computational kernels from many non-graphical applications.

In this paper, we present a correlation-based stereo algorithm that is completely implemented on the GPU. Many advanced features such as adaptive window and cross-checking are included in our implementation. Compared with previous approaches that use GPU accelerations [25, 4], our optimized implementation achieves a significant speed boost even with the same type of graphics hardware. In addition, we have measured the accuracy of our approach using the widely used ground truth data from Scharstein and Szeliski [20]. When real-world images are used, our approach compares favorably with several non real-time methods.

[1]The number of disparity evaluations per seconds corresponds to the product of the number of pixels times the disparity range times the obtained frame-rate and therefore captures the performance of a stereo algorithm in a single number.

## 2 Related Work

In this section, we first present an overview of stereo algorithms, in particular, real-time ones. Then, for motivation and clarity, we explain the basic architecture of modern GPUs.

### 2.1 Stereo Reconstruction

Stereo vision is one of the oldest and most active research topics in computer vision. It is beyond the scope of this paper to provide a comprehensive survey. Interested readers are referred to a recent survey and evaluation by Scharstein and Szeliski [21]. While many stereo algorithms obtain high-quality results by performing optimizations, today only correlation-based stereo algorithms are able to provide a dense (per pixel) depth map in real time on standard computer hardware.

Only a few years ago even correlation-based stereo algorithms were out of reach of standard computers so that special hardware had to be used to achieve real-time performance [8, 12, 23, 13, 6].

In the meantime, with the tremendous advances in computer hardware, software-only real-time systems begin to merge. For example, Mulligan and Daniilidis proposed a new trinocular stereo algorithm in software [16] to achieve 3-4 frames/second on a single multi-processor PC. Hirschmuler introduced a variable-window approach while maintaining real-time suitability [10, 9]. Commercial solutions are also available. The stereo algorithm from Point Grey Research [11] yields approximately 80Mde/s on a 2.8Ghz processor, at 100% utilization.

All these methods used a number of techniques to accelerate the calculation, most importantly, assembly level instruction optimization using Intel's MMX extension. While the reported performance is sufficient to obtain dense-correspondences in real-time, there are few CPU cycles left to perform other tasks including high-level interpretation of the stereo results.

Recently, Yang et al [26] proposed a completely different approach. They presented a real-time multi-baseline system that takes advantage of commodity graphics hardware. The system was mostly aimed at novel view generation, but could also return depth values. The approach used the programmability of modern graphics hardware to accelerate the computation. But it was limited to use a $1 \times 1$ correlation window. Later, Yang and Pollefeys [25] introduced a pyramid-shaped correlation kernel that strikes a balance between large windows (more system errors) and small windows (more ambiguities), and can very efficiently be evaluated on graphics hardware. Almost around the same time, Zach et al. introduced a mesh-based stereo algorithm that lends itself well on commodity graphics hardware [4].

The method we propose in this paper is most related to these techniques. Based on [25], we introduce a number of improvements and optimizations, such as accurate evaluation of matching costs, adaptive window, cross-checking, multiple-disparity packing, to improve both the accuracy and speed of stereo reconstruction.

### 2.2 A Brief Review of Modern Graphics Hardware


Figure 1: Rendering Pipeline

GPUs are dedicated processors designed specifically to handle the intense computational requirements of display graphics, i.e., rendering texts or images over 30 frames per second. As depicted in Figure 1, a modern GPU can be abstracted as a rendering pipeline for 3D computer graphics (2D graphics is just a special case) [22].

The inputs to the pipeline are geometric primitives, i.e., points, lines, polygons; and the output is the *framebuffer*–a two-dimensional array of pixels that will be displayed on screen.

The first stage operates on geometric primitives described by vertices. In this *vertex-processing* stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage, *rasterization*. The rasterizer produces a series of framebuffer addresses and color values, each is called a *fragment* that represents a portion of a primitive that corresponds to a pixel in the framebuffer.

Each fragment is fed to the next *fragment processing* stage before it finally alters the framebuffer. Operations in this stage include texture mapping, depth test, alpha blending, etc.

Until a few years ago, commercial GPUs, such as the RealityEngine from SGI [2], implement in hardware a fixed rendering pipeline with configurable parameters. As a result their applications are restricted to graphical computations. Driven by the market demand for better realism, the most recent generation of commercial GPUs such as the NVIDIA GeForce FX [18] and the ATI Radeon 9800 [3] added significant programmable functionalities in both the vertex and the fragment processing stage(see Figure 1). They allow developers to write a sequence of instructions to modify the vertex or fragment output. These programs are directly executed on the GPUs to achieve comparable performance to fixed-function GPUs. For example, the NVIDIA GeForce FX series can reach a peak performance of 6 Gflops in the vertex processor and 21 Gflops in the fragment processor [15].

Many researchers, including us, have recognized the computation power of GPUs for *non-graphical* applications. Interested readers are referred to `http://www.gpgpu.org` for a collection of examples of applications successfully implemented on the GPU.

## 3 Method

Given a pair of images, the goal of a stereo algorithm is to establish pixel correspondences between the two images. The correspondence can be expressed in general as a disparity vector, i.e., if $P_L(x, y)$ and $P_R(x', y')$ are corresponding pixels in the left and right image respectively, then the disparity of $P_L(x, y)$ and $P_R(x', y')$ is defined as the difference of their image coordinates–$[x - x', y - y']$. Therefore, the output of a stereo algorithm is a disparity map, i.e., a map that records the disparity vector for every pixel in one image (the reference image) – the disparity map for the other image is automatically defined because of the symmetry in disparity vectors.

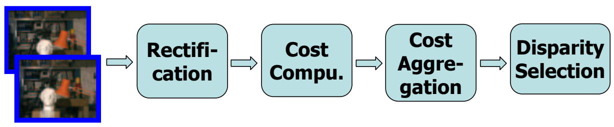


Figure 2: Block diagram of our stereo algorithm

Illustrated in Figure 2, our algorithm contains four major steps: rectification, matching cost computation, cost aggregation, and finally disparity selection. Rectification involves a 2D projective transformation for each image so that the epipolar lines are aligned with scan lines. In this case, the disparity vector degrades to a scalar since corresponding pixels must be on the same scan line, i.e., $y \equiv y'$. We choose to work with rectified images since it brings a number of performance advantages (we will discuss more later). In the second step, a matching cost for every possible disparity value for each pixel is computed. To reduce the ambiguity in matching, the cost is summed over a small neighboring window (support region) in the third aggregation step. The implicit assumption made here is that the surface is locally smooth and frontal-parallel (facing the camera), so neighboring pixels are likely to have the same disparity value. In the last disparity selection step, we use a "winner-take-all" strategy: simply assign each pixel to the disparity value with the minimum cost.

While our algorithm resembles a classic stereo vision algorithm, implementing it efficiently on a GPU is challenging because of GPU's unique programming model. In the next few sections, we will discuss how to map these steps on graphics hardware to receive maximum acceleration.

## 3.1 Rectification

The standard approach to perform image-pair rectification consist of applying $3 \times 3$ homographies to the stereo images that will align epipolar lines with corresponding scanlines [7]. This can be efficiently implemented as a projective texture mapping on a GPU. It is also a common practice to correct lens distortions at the same time. Unlike rectification, dealing with lens distortions requires a non-linear transformation. A common optimization is to create a look-up table that encodes the per-pixel offset resulting from lens distortion correction and the rectifying homography. The latest generation of graphics hardware supports dependent-texture look-up that makes precise per-pixel correction possible. With older graphics hardware, this warping can be approximated by using a tesselated triangular mesh. This type of approach would also allows to use more advanced non-linear rectification transformations that can be necessary of the epipoles are in (or close to) the images [19].

## 3.2 Matching cost computation

A widely used matching cost is the the absolute difference between the left and right pixel intensities:

$$|I_L(x, y) - I_R(x + d, y)| \ . \tag{1}$$

where $d$ is the hypothesized disparity value. Under the Lambertian surface assumption, a pair of corresponding pixels in the left and right view should have identical intensities, leading to a zero(optimal) matching cost.

Since the images are rectified, every disparity value corresponds to a horizontal shift in one of the images. In our implementation, we store the two input images as two textures. For each disparity hypothesis $d$, we draw a screen-sized rectangle with two input textures, one of them being shifted by $d$ pixels. We use the fragment program to compute the per-pixel absolute difference, which is written to the framebuffer. The absolute difference (AD) image is then transferred to a texture, making the framebuffer ready for the matching cost from a different disparity value. To search over $N$ disparity hypothesis, $N$ rendering passes are needed.

In this baseline implementation, there are several places that can be improved using advanced features available in the newer generation of GPUs.

First is the copy from framebuffer to texture. This can eliminated by using the *P-buffer* extension [1]. P-buffer is a user-allocated off-screen buffer for fragment output. Unlike the framebuffer, it can be used directly as a texture. In our implementation, we create one or more P-buffers depending on the disparity search range. Each P-buffer should be as large as possible so that multiple AD images can be stored in a single P-buffer to reduce the switching overhead.

Another optimization is to use the vector processing capability of graphics hardware. One possibility is to pre-pack the input images into the four channels of textures. Both images are first converted into gray-scale ones (if they are color); then they are replicated into all four channels of the corresponding texture, but one of them (say the right one) is shifted incrementally in each channel, i.e., the red channel stores the original right image, the green channel stores the original right image horizontally shifted by one pixel, so on and so forth. With these packed images, we can compute the matching costs for four consecutive disparity values in a single pass. But this approach discards the color information, we instead implemented a quite complicated fragment program to compute the matching costs over four disparity values in a single pass. It essentially retrieves one pixel from the reference image, and four pixels from the other image that correspond to disparity values of $d$ to $d + 3$. Then four AD values are calculated and packed into one RGBA fragment output. Since these operations can be pipelined, we noticed little performance degradation compared to the pre-packing approach.

## 3.3 Cost Aggregation

While it is possible to assign disparity values directly based on the per-pixel difference values from multiple images [14, 26], it is necessary to use larger support region in the stereo case with only two input images.

Stereo algorithms typically sum the matching cost over a small window to increase the robustness to noise and texture variation. However, choosing the size of the aggregation window is a difficult problem. The probability of a mismatch goes down as the size of the window increases [17]. However, using large windows leads to a loss of accuracy and to the possibility of missing some important image features. This is especially so when large windows are placed over occluding boundaries. This problem is typically dealt with by using a hierarchical approach [8], or by using special approaches to deal with depth discontinuities [10].

More recently, Yang and Pollefeys introduced a different approach that is better suited to the implementation on a GPU. Their goal was to combine the global characteristics of the large windows with the well-localized minima of the small windows. They achieved this by adding up the aggregated matching costs over differently-sized windows.

Modern GPUs have built-in box-filters to efficiently generate all the mipmap levels needed for texturing. Starting from a base image $P^0$ the following filter is recursively applied:

$$P^{i+1}_{u,v} = \frac{1}{4} \sum_{q=2v}^{2v+1} \sum_{p=2u}^{2u+1} P^i_{p,q},$$

where $(u, v)$ and $(p, q)$ are pixel coordinates. Therefore, it is very efficient to sum values over $2^n \times 2^n$ windows. Note that at each iteration of the filter the image size is divided by two. Therefore, a disadvantage of this approach is that the cost summation can only be evaluated exactly at every $2^n \times 2^n$ pixel location. For other pixels, approximate values can only be obtained by interpolation.

We choose to use an adaptive window that can be accurately evaluated at every pixel location. Note by enabling bilinear texture interpolation and sampling in the middle of 4 pixels, it is possible to average those pixels. To sum over a large window, we implement a two-pass algorithm. In the first pass, we draw every AD image with orthographic projection and a fragment program is implemented to sample and sum the AD image at four different locations per pixel (shown in Figure 3(a)); this is equivalent to sum over a $4\times4$ window. The resulting sum-of-absolute-difference (SAD) image is stored in another P-buffer and used as a texture for

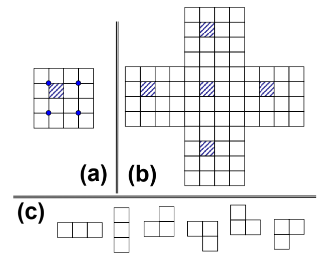


Figure 3: Adaptive window for cost aggregation. (a) sum the cost over a $4 \times 4$ windows with four bilinearly interpolated values (sampled at the circle locations). (b) in the second pass, four more SAD values are sampled and the smaller two are added to the SAD score of the current pixel. Therefore a total of six support windows is possible, shown in (c).

the second pass, in which the four neighbors of each pixel are sampled. As shown in Figure 3(b), these four neighbors are $(u-4, v)$, $(u+4, v)$, $(u, v+4)$, and $(u, v-4)$. Their values (SAD scores) are sorted, and the smaller two are added to $P(u, v)$ as the final matching cost. All these operations are implemented in a fragment program.

Our adaptive scheme has six different support windows, each corresponding to a different shape configuration–corner, edge, etc(Figure 3(c)). The one with the minimum score is used as the aggregated matching cost.

## 3.4 Disparity Selection

Typical in real-time stereo algorithms, we use a "winner-take-all" strategy that assigns each pixel to the disparity value with the minimum cost. This step in fact can be combined with the previous aggregation step. Once a pixel's matching cost at a certain disparity is computed, it is sent to the framebuffer as a depth value while the disparity value is encoded as the color. In our implementation, we draw each SAD image sequentially. By enabling the depth test, each pixel in the final framebuffer will be assigned the color value (disparity) with the minimum depth (matching cost). That concludes the stereo computation.

When dealing with packed SAD images, we have to implement a fragment program that finds out the minimum value among the four channels and compute the corresponding color value.

**Cross-Checking** So far, we have calculated a disparity map using one image as the reference. We can apply the same algorithm with the other image as the reference. This will yield another disparity map. These two maps may not be identical due to issues such as occlusions and sampling. We can therefore remove the inconsistent disparity values to increase the accuracy. This process is called *cross check-*

*ing* [5]. Working with rectified images, it is quite easy to efficiently implement cross-checking. As shown in Figure 4, the SAD images (each corresponding a single disparity hypothesis) are aligned with the reference image, therefore different matching costs for a pixel in the reference image are aligned in a column in the disparity direction. In the meantime, different matching costs for a pixel in the *other* image are aligned in a diagonal direction. Thus, we just need to draw the SAD images with an incremental horizontal shift to calculate the second disparity map. The two disparity maps are copied to textures and compared through a fragment program. Pixels with inconsistent disparities are removed.

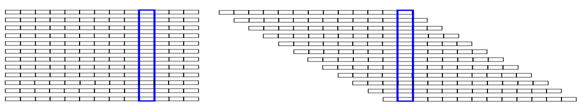

Figure 4: Cross-checking with rectified images.

## 3.5 Summary of Implementation

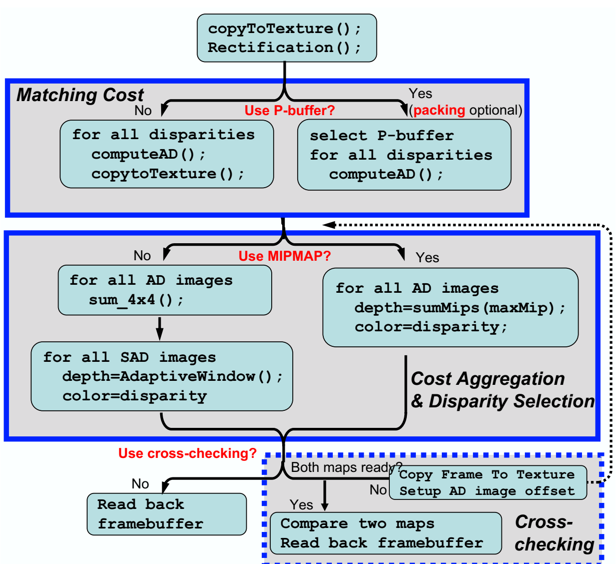


Figure 5: A block diagram of our implementation.

We summarize our implementation in Figure 5. Input images are first sent to the graphics board as textures. Then in the display routine, we usually draw screen-sized rectangles with orthographic projection. The rasterized rectangles together with different textures are fed into the fragment processors, in which the majority of the calculations, including rectification, absolute difference, and cost aggregation, is carried out on a per-pixel basis. Since a modern GPU typically has multiple fragment processors that work in parallel, the calculation is greatly accelerated.

# 4 Results

We have implemented our proposed method in OpenGL, the complete sample code is available in [24]. In this section, we will present some quantitative results both in accuracy and speed.

For accuracy evaluation, we use the data set from the Middlebury stereo evaluation page [20]. There are four stereo pairs, each with a ground truth disparity map. We calculate disparity maps using our method and compare them with the ground truth. The result is summarized in Table 1, and some disparity maps are shown in Figure 6 and 7. The "All" columns show the overall error rates, which is calculated as follows: If a pixel's disparity differs more than one from the ground truth, it is considered as a bad match. The error rate is the ratio between the total number of bad matches and total number of pixels, excluding the boundary pixels (which are also marked in the ground truth data). The Middlebury page uses the same accuracy measure [20]. For results after cross-checking, we compute the error rate as the ratio of incorrectly matched pixels and pixels with disparity values, excluding the boundary and occluded pixels as usual. In these cases, we also calculate the percentage of "missing" pixels. These numbers are displayed in the "Miss" columns.

| Alg | **Tsukuba** | | **Sawtooth** | | **Venus** | | **Map** | |
|---|---|---|---|---|---|---|---|---|
| | All | Miss | All | Miss | All | Miss | All | Miss |
| MIP | 7.07 | 0 | 10.4 | 0 | 13.3 | 0 | 2.33 | 0 |
| AW4 | 9.68 | 0 | 5.79 | 0 | 15.7 | 0 | 0.91 | 0 |
| MPX | 2.96 | 22.5 | 6.76 | 13.1 | 4.96 | 16.9 | 0.69 | 12.7 |
| AWX | 3.33 | 28.5 | 4.02 | 19.1 | 2.46 | 35.6 | 0.80 | 22.2 |

Table 1: Reconstruction Accuracy. All numbers are in percentage. "All" is the overall error rate. "Miss" is the percentage of pixels with undefined disparity values due to the inconsistency from cross-checking. Four different algorithms are tested; they are the mipmap method (MIP), the adaptive window method (AW4), and their derivations with cross-checking (MPX and AWX).

Several methods are tested. They are the mipmap method (MIP) that is introduced in [25], the adaptive window method (AW4), and their derivations with cross-checking (MPX and AWX). The number of mipmap level used in the MIP method is set to six cross all tests, and the AW4 method has no parameter. Looking into the results, we can find several interesting observations. First, the AW4 method does preserve depth continuity much better

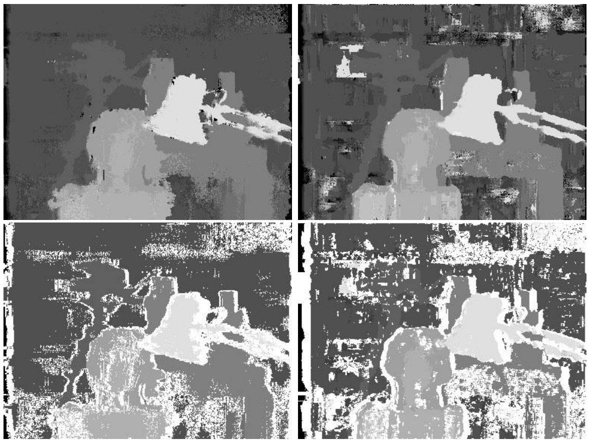

Figure 6: Estimated disparity maps from the Tsukuba set. Methods used are MIP, AW4, MPX, AWX (from left to right, top to down). Pure white in maps resulting from cross-checking indicates missing pixels.

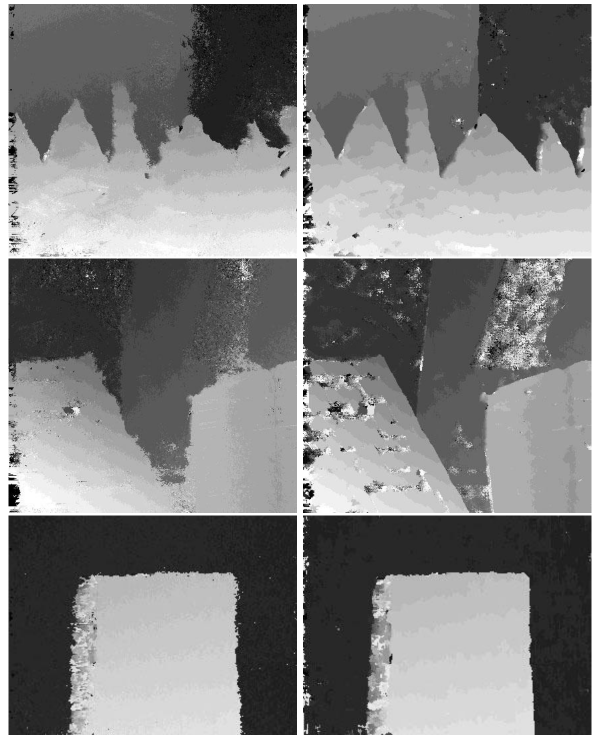

Figure 7: Estimated disparity maps from other data sets. Images on the left are computed with the MIP method, while these on the right are from the AW4 method.

than the mipmap method (see Figure 7), but the overall error rates are similar. Secondly, cross-checking substantially reduces the error rate by half or more, but in the meantime causes many pixels with no disparity value. Thirdly, while the results from real images (Tsukuba and Map) are within the expectation of a local correlation-based algorithm and better than several non-realtime methods (see [20]), the results from the remaining synthetics images are substantially worth than those listed on the Middlebury page. We were initially puzzled by this outcome but we now believe it is due to the lack of precision in the AD image since the matching cost is stored as a unsigned character. This can be improved by using floating point textures. However in real applications, the image noise probably outweighs the inaccuracy caused by storing the matching cost as an unsigned character.

In term of speed, we test our implementation on an ATI Radeon 9800 XT card with 256 MB of graphics memory. The card is housed in a 2.8 Ghz PC with 512 MB of main memory. We experimented with five methods, MIPMAP (MIP), adaptive window (AW4), and MIPMAP stored in P-buffer (MPB), MIPMAP with packed AD images in P-buffer (MPP), and MIPMAP stored in P-buffer with cross-checking (MPB_X). The MIPMAP summation level is set to six. The performance data is summarized in Table 2 with the first two rows showing various overheads. For each method, the time to calculate a disparity map for different size input and disparity range is displayed. These numbers do not include the overhead, but we do include the overhead to calculate the throughput: million disparity evaluation per second (Mde/s).

As we can see in Table 2, our implementation can reach 289 Mde/s, which is achieved by using P-Buffer and packed AD images. This performance compares favorably with software stereo implementations, such as the package from Point Grey Research [11] with an estimated 80 Mde/s on a 2.8Ghz PC. In addition, we still have the majority of the CPU cycles available for other tasks since our approach runs on a GPU. There are also a few numbers listed as “not available” because of the memory limitation in the graphics hardware–it cant allocate enough P-buffers to store all the AD images.

# 5 Conclusion

We have introduced techniques to implement a *complete* stereo algorithm on commodity graphics hardware. Compared to previous approaches using GPUs [25, 4], our approach includes several major improvements, such as accurate evaluation of matching costs, adaptive window, and cross-checking. Thanks to rapid advancement in graphics hardware and careful algorithm design, all the calculations are performed by the GPU, avoiding the GPU-CPU communication bottleneck as in [4]. Performance tests have shown that our implementation running on an ATI Radeon 9800 graphics card can calculate up to 289 million disparity evaluations per second.

| Overhead | size | Download (ms) | Read-back (ms) | Rectification (ms) |
|---|---|---|---|---|
| | 512 | $1.12 \times 2$ | 6.25 | $3.2 \times 2$ |
| | 256 | $0.29 \times 2$ | 1.62 | $0.6 \times 2$ |

| Size | Disp. Range | MIP (ms) | MIP (Mde/s) | AW4 (ms) | AW4 (Mde/s) | MPB (ms) | MPB (Mde/s) | MPP (ms) | MPP (Mde/s) | MPB_X (ms) | MPB_X (Mde/s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $512^2$ | 16 | 24 | 108 | 33.8 | 86 | 19.5 | 122 | 15.6 | 138 | 31.3 | 182 |
| | 32 | 47.7 | 134 | 67.1 | 102 | 37.7 | 159 | 28.9 | 192 | 59.8 | 225 |
| | 64 | 94.9 | 153 | 133.6 | 113 | n/a | n/a | 55.2 | 239 | n/a | n/a |
| | 94 | 141.9 | 161 | 199.7 | 117.3 | n/a | n/a | 72.1 | **289** | n/a | n/a |
| $256^2$ | 16 | 8.5 | 88 | 9 | 84 | 7 | 101 | 5.7 | 115 | 9.8 | 158 |
| | 32 | 16.8 | 104 | 17.7 | 99 | 10.8 | 148 | 9 | 169 | 16.4 | 212 |
| | 64 | 33.5 | 114 | 35.2 | 109 | 18.6 | 191 | 15.8 | 218 | 29.6 | 254 |
| | 96 | 50.1 | 118 | 52.5 | 113 | 28.3 | 198 | 22.4 | 243 | 44.2 | 264 |

Table 2: Performance on an ATI Radeon 9800 card. The maximum mipmap level is set to six in all MIPMAP-based tests. The time per reconstruction does not include the overhead, while the calculation for the million disparity evaluations/second (Mde/s) does.

Looking into the feature, we are looking at ways to efficiently implement more advanced reconstruction algorithms on graphics hardware. This work will be eased with newer generations of graphics hardware providing more and more programmability. We also hope that our method inspires further thinking and additional new methods to explore the full potentials of GPUs for real-time vision.

# Acknowledgments

The authors would like to thank ATI and NVIDIA for their generous hardware donations and technical support. The work is supported in part by fund from the office of research at the University of Kentucky and NSF grant IIS-0313047.

# References

[1] OpenGL Specification 1.4, August 2003. http://www.opengl.org/documentation/specs/ version1.4/glspec14.pdf.

[2] K. Akeley. Realityengine graphics. In *Proceedings of SIGGRAPH*, 1993.

[3] ATI Technologies Inc. ATI Radeon 9800, 2003. http://www.ati.com/products/radeon9800.

[4] C. Zach C., A. Klaus, and K. Karner. Accurate Dense Stereo Reconstruction using Graphics Hardware. In *EUROGRAPHICS 2003*, pages 227–234, 2003.

[5] S. D. Cochran and G. Medioni. 3D Surface Description from Binocular Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 14(10):981–994, 1992.

[6] A. Darabiha, J. Rose, and W. J. MacLean. Video-Rate Stereo Depth Measurement on Programmable Hardware. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 203–210, 2003.

[7] O. Faugeras. *Three-Dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, 1993.

[8] O. Faugeras, B. Hotz, H. Mathieu, T. Viville, Z. Zhang, P. Fua, E. Thron, L. Moll, G. Berry, J. Vuillemin, P. Bertin, and C. Proy. Real time sorrelation-based stereo: Algorithm, amplementations and application. Technical Report 2013, INRIA, August 1993.

[9] Heiko Hirschmuler. Improvements in Real-Time Correlation-Based Stereo Vision. In *Proceedings of IEEE Workshop on Stereo and Multi-Baseline Vision*, pages 141–148, December 2001.

[10] H. Hirschmuller, P. Innocent, and J. Garibaldi. Real-Time Correlation-Based Stereo Vision with Reduced Border Errors. *International Journal of Computer Vision*, 47(1-3), April-June 2002.

[11] Point Grey Research Inc. http://www.ptgrey.com.

[12] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A Stereo Engine for Video-rate Dense Depth Mapping and Its New Applications. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, pages 196–202, June 1996.

[13] K. Konolige. Small Vision Systems: Hardware and Implementation. In *Proceedings of the 8th International Symposium in Robotic Research*, pages 203–212. Springer-Verlag, 1997.

[14] K. Kutulakos and S. M. Seitz. A Theory of Shape by Space Carving. *International Journal of Computer Vision (IJCV),*, 38(3):199–218, 2000.

[15] K. Moreland and E. Angel. The FFT on a GPU. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings*, pages 112–119, 2003.

[16] J. Mulligan, V. Isler, and K. Daniilidis. Trinocular Stereo: A New Algorithm and its Evaluation. *International Journal of Computer Vision (IJCV), Special Issue on Stereo and Multi-baseline Vision*, 47:51–61, 2002.

[17] H. Nishihara. PRISM, a Pratical Real-Time Imaging Stereo Matcher. Technical Report A.I. Memo 780, MIT, 1984.

[18] NVIDIA Corporation. GeForce FX, 2003. http://www.nvidia.com/page/fx desktop.html.

[19] M. Pollefeys, R. Koch, and L. Van Gool. A Simple and Efficient Rectification Method for General Motion. In *Proceedings of International Conference on Computer Vision (ICCV)*, pages 496–501, Corfu, Greece, 1999.

[20] D. Scharstein and R. Szeliski. www.middlebury.edu/stereo.

[21] D. Scharstein and R. Szeliski. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision*, 47(1):7–42, May 2002.

[22] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 1.3), 2001. http://www.opengl.org.

[23] John Woodfill and Brian Von Herzen. Real-Time Stereo Vision on the PARTS Reconfigurable Computer. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 201–210, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[24] R. Yang. http://galaga.netlab.uky.edu/ ryang/research/ViewSyn/ realtime.htm.

[25] R. Yang and M. Pollefeys. Multi-Resolution Real-Time Stereo on Commodity Graphics Hardware. In *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 211–218, 2003.

[26] R. Yang, G. Welch, and G. Bisop. Real-Time Consensus-Based Scene Reconstruction Using Commodity Graphics Hardware. In *Proceedings of Pacific Graphics 2002*, pages 225–234, Beijing, China, October 2002.