Efficient Computation of Histograms on the GPU

Alexander Kubias*
University of Koblenz-Landau

Frank Deinzer†
Siemens Medical Solutions

Matthias Kreiser‡
Siemens Medical Solutions

Dietrich Paulus§
University of Koblenz-Landau

Abstract

Recently, several image processing algorithms have been ported on the GPU because of its fast increase of performance. Some of these algorithms need to compute a histogram, but the computation of the histogram is not well supported on the GPU because of its architecture. To cope with this problem, we present a new method for the efficient computation of histograms on the GPU. Therefore, we use a fragment shader, occlusion queries and the depth-buffer. The gray level values of the image of which the histogram will be calculated are taken as depth values. The depth-buffer in combination with occlusion queries can efficiently be used to count the number of gray values that fall into each histogram bin. We will show in our experiments that our method is up to ten times faster than an existing technique for the GPU-based computation of histograms. For a small number of bins in the histogram or for large images, our method is even faster than a CPU implementation.

CR Categories: D.1.0 [Software]: Programming Techniques—General

Keywords: image processing, histogram, GPU, GPGPU

1 Introduction and Related Work

Graphics hardware has become an attractive target for image processing applications because of its rapid increase of performance and because of recent improvements of its programmability [Owens et al. 2005].

For instance, several algorithms for image segmentation have been accelerated on graphics cards. In [Viola et al. 2003] a threshold-based 3D segmentation was performed on the GPU that was 8 times faster than a CPU implementation. Level set segmentation has also been implemented on GPUs [Rumpf and Strzodka 2001].

Furthermore, graphics hardware has been used to accelerate computer vision applications. Fung et al. exploited graphics hardware in order to speed up a camera-based head-tracking system [Fung and Mann 2004]. In [Yang and Pollefeys 2005] a system for real-time stereo depth extraction from multiple images has been established on the GPU. The open-source computer vision library OpenVIDIA [Fung and Mann 2005] provides a graphics hardware accelerated processing framework for computer vision. In OpenVIDIA several algorithms for edge detection, filtering, feature identifying and matching are implemented on the GPU.

The GPU has recently been applied for image registration purposes. In [Strzodka et al. 2004] the regularized gradient flow algorithm has been implemented on the GPU for non-rigid 2D/2D-registration. This algorithm has been extended in [Köhn et al. 2006] for 3D/3D-registration, likewise by using the GPU. Several approaches were developed to perform the 2D/3D-registration on the GPU. The 2D/3D-registration consists of two main parts, namely the generation of digitally reconstructed radiographs (DRRs) and the computation of similarity measures [Penney 1999]. Some of these approaches like [LaRose 2001] and [Khamene 2005] generated only the DRRs on the GPU, while the similarity measures were still computed on the CPU. Lately, some approaches like [Wein 2003] implemented both parts of the 2D/3D-registration on the GPU and achieved huge performance speedups. All these publications have in common that the algorithmic problems were ported on the GPU and, thus, they could be efficiently solved using the possibilities of the GPU.

Another important operation for image processing is the computation of histograms. A histogram provides information about how often a certain gray level value or a certain color value appears in the image. For instance, histograms are needed for histogram equalization and to determine the contrast and the brightness of an image. Furthermore, histograms are the fundament for the computation of histogram-based similarity measures such as Mutual Information [Collignon et al. 1995] and Joint Entropy [Hill et al. 1994].

In the past, histograms were usually computed on the CPU. Therefore, one had to run over the image in a loop and had to count how often a certain value occurs. As a result, one received the absolute number of each gray level value in the image. In the recent past, new approaches arose which were able to compute histograms on the GPU. This was necessary because many image processing algorithms had been ported on the GPU and some of these algorithms are based on histograms. In these scenarios it is not feasible to compute the histogram on the CPU because it is too time-consuming to download the large images from the GPU to the CPU because of the bottleneck between the CPU and the GPU.

If the image or the texture is already on the GPU, it is more efficient to compute the histogram directly on the GPU. Therefore, some new techniques have been presented. Some of these techniques like [Green 2005] use the so-called occlusion queries that count how many fragments have passed the rendering step. In [Green 2005] the fragment shader gets the image as input for which the histogram will be computed. For an histogram with n_{bins} bins the fragment shader must be executed n_{bins} times. In the first iteration all pixels are counted that belong to the lowest bin. If a pixel has a gray level value that is in this bin, the fragment can pass. Thus, it will be counted by the occlusion query. Otherwise, if the pixel is outside the given range, the fragment shader will discard this pixel, so that it will not be counted by the occlusion query. After repeating the procedure in n_{bins} rendering steps for n_{bins} bins, one retrieves the histogram of absolute values. But this technique has a lack of performance. It can only be utilized if the image for that the

*e-mail: kubias@uni-koblenz.de
†e-mail: frank.deinzer@siemens.com
‡e-mail: matthias.kreiser@siemens.com
§e-mail: paulus@uni-koblenz.de

histogram should be computed is very small or if the histogram has only a small number of bins.

Another technique that was presented in [Deuerling-Zheng et al. 2006] uses alpha blending in order to compute histograms on the GPU. In this implementation each pixel is mapped to a single vertex. Depending on the gray level value of the regarded vertex, the vertex is scattered to that position that stands for this certain gray level value. After the rendering the final color of a pixel on the screen is a mixture of all fragments which have the same position, if alpha-blending is enabled. If all the input pixels have the same intensity and the alpha-blending function is set as additive, the frequency of the input pixels can be retrieved from the final intensity of the output pixel.

In [Fluck et al. 2006] a further method for computing histograms on the GPU is described. In this method the image of which the histogram should be calculated is divided into several tiles. In each tile the occurrence of a certain gray level value is counted and stored in one particular texel of this tile. Then, the tiles are summed up to a global histogram by using a texture reduce operation.

In addition, there is an OpenGL extension called glGetHistogram. This extension does not work very efficiently because it downloads the texture from the GPU to the CPU and computes the histograms on the CPU. Because of its bad performance this extension is rarely used.

In order to have a possibility to compute histograms on the GPU more efficiently, we will present a new method for computing histograms on the GPU. In section 2 we describe our method formally and how it is implemented on the GPU. In section 3 our method is compared to other approaches. Finally, in section 4 we give a conclusion of our method and we provide information about our future work.

2 Efficient Computation of Histograms

2.1 Histograms on the CPU

In the context of image processing one is often interested in the histogram of an image. A histogram is a function that assigns a gray level value to its relative or absolute frequency. So the likelihood is known that a certain gray level value occurs.

In order to compute the (not cumulative) histogram of absolute frequencies h^{abs}, one has to count how often a certain gray level occurs. Then, the histogram of relative frequencies h^{rel} is computed from the histogram of absolute frequencies h^{abs} by dividing the absolute frequency by the total number of pixels n_{pixels} in the image.

For a gray level image with 256 gray levels (i.e. $n_{\text{bins}} = 256$ distinct bins), the histogram of relative frequencies h^{rel} assigns each gray level value between 0 and 255 a relative frequency between 0 and 1, as shown in (1).

$$h_i^{\text{rel}} \in [0;1] \quad \text{with} \quad \sum_i h_i^{\text{rel}} = 1 \quad \text{and} \quad i \in [0;255] \tag{1}$$

A cumulative histogram of relative frequencies H^{rel} is derived from h^{rel} by (2). A cumulative histogram of absolute frequencies H^{abs} is calculated by (3).

$$H_k^{\text{rel}} = \sum_{i=0}^{k} h_i^{\text{rel}} \tag{2}$$

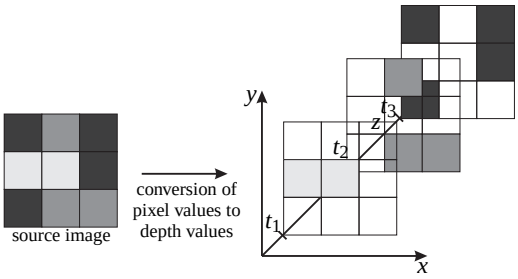

Figure 1: Converting the gray level values of the image into depth values

$$H_k^{\text{abs}} = \sum_{i=0}^{k} h_i^{\text{abs}} \tag{3}$$

2.2 Depth-Buffer (z-Buffer) Based Computation of Histograms

In our approach we want to use the potentials of the depth-buffer in order to compute histograms. Because of that, the gray level values of the image, of which the histogram will be calculated, are taken as depth values. Thus, a depth value v_{depth} is assigned to each pixel, depending on its gray level value v_{gray} using (4). In (4) v_{gray} has a normalized value between 0 and 1.

$$v_{\text{depth}} = 1 - v_{\text{gray}} \tag{4}$$

If its gray level value is 0 (black pixel), the depth value 1 is assigned, which means that the pixel is far away from the viewer. The depth value 0 is assigned to a pixel, if its gray level value is 1 (white pixel). The depth value 0 means that the pixel is very near to the viewer. An example is shown in Figure 1, where an image with 3×3 pixels and three different gray level values is converted into depth values.

After converting the gray level values of the image into depth values, the depth-buffer is filled with a depth value for each pixel position. Now n_{bins} planes parallel to the viewing plane have to be drawn, if we want to distinguish n_{bins} gray level values. Each plane must have the same size as the input image and there must be a constant distance between the planes. Furthermore, the projection must be orthogonal and, thus, no perspective distortion occurs.

The first plane is drawn in such a way that its own pixels would retrieve a depth value of $1 - \frac{1}{n_{\text{bins}}}$. When the first plane is being drawn, the depth-buffer checks for each pixel position if there is already a depth value that is smaller or equal. If there is a depth value for this pixel position that is smaller or equal, this pixel of the first plane will not be drawn. Otherwise, the pixel of the first plane will be drawn and will be counted. By counting the number of drawn pixels, one also counts the number of pixels that have a gray level value of 0. In Figure 2 the situation is depicted, when the first plane has been drawn and the number of drawn pixels of the first plane has been counted.

The second plane is drawn in a distance of $\frac{1}{n_{\text{bins}}}$ to the first plane. So its pixels will get a depth value of $1 - \frac{2}{n_{\text{bins}}}$. The situation after drawing the second plane is illustrated in Figure 3. By counting the number of drawn pixels of the second plane, one counts the

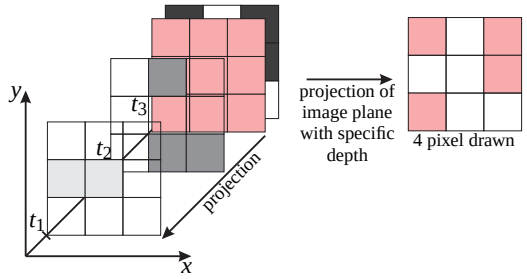

Figure 2: The first plane has been drawn and the number of drawn pixels of the first plane has been counted.

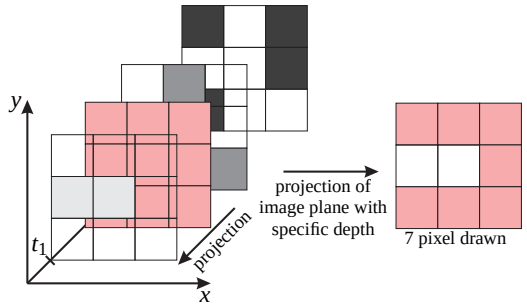

Figure 3: The second plane has been drawn and the number of drawn pixels of the second plane has been counted.

number of pixels in the image that have a gray level value of $\frac{1}{n_{\text{bins}}-1}$ or smaller.

The pixels of the last plane will get a depth value of 0. As one can see, the planes are drawn from the back to the front.

When all n_{bins} planes were drawn, one immediately retrieves the cumulative histogram of absolute frequencies H^{abs}. In order to obtain the histogram of absolute frequencies h^{abs}, h_i^{abs} can be calculated from H_i^{abs} according to (5).

$$h_i^{\text{abs}} = H_i^{\text{abs}} - H_{i-1}^{\text{abs}} \quad \text{with} \quad i \in [0;255] \tag{5}$$

Finally, the histogram of relative frequencies h^{rel} can be calculated from h_i^{abs} according to

$$h_i^{\text{rel}} = \frac{h_i^{\text{abs}}}{n_{\text{pixels}}} \quad \text{with} \quad i \in [0;255] \quad . \tag{6}$$

2.3 Implementation

In order to compute the histogram efficiently, we use an algorithm that is executed on the GPU. The gray level image, of which the histogram should be calculated, is given to the algorithm as a texture. In our method we use a fragment shader, occlusion queries and the depth-buffer.

First, the gray level value v_{gray} of each pixel of the input image must be transformed according to (4) into a depth value v_{depth} that is stored in the depth-buffer. For this transformation a fragment shader is executed that performs this assignment of gray level values to depth values. The resulting fragment shader is shown in the following. It is important to know that the gray level values in the texture are already normalized.

```
void main(in float2 coords : TEXCOORD0,
    uniform sampler2D texture0,
    out float depth : DEPTH)
{
    float4 x = tex2D(texture0, coords);
    depth = (1 - x.x);
}
```

Afterwards, n_{bins} rendering passes are following, if we want to distinguish n_{bins} gray level values. In each rendering step a new plane parallel to the viewing plane has to be drawn and the number of drawn pixels of the current plane has to be counted. Every new plane is drawn and translated into the correct depth using the subsequent source code.

```
void drawplane(float k)
{
    glTranslatef(0.0, 0.0, k);
    glBegin(GL_QUADS);
        glVertex3f (-1, -1, 0);
        glVertex3f ( 1, -1, 0);
        glVertex3f ( 1,  1, 0);
        glVertex3f (-1,  1, 0);
    glEnd();
}
```

In each rendering pass the graphic card decides by means of the depth-buffer which pixel of the new plane is drawn. A pixel is only drawn if its depth value is smaller than the corresponding value in the depth buffer. In order to count the number of drawn pixels, we use the so-called occlusion queries. These must be initialized before they can be used. In the following source code the initialization of 256 occlusion queries is performed.

```
GLuint query[256];
glGenQueriesARB(256, query);
```

After the initialization each rendering pass uses one occlusion query to count the number of drawn pixels.

```
int nrDrawnPixels = 0;
glBeginQueryARB(GL_SAMPLES_PASSED_ARB, query[0]);
drawplane(k);
glEndQueryARB(GL_SAMPLES_PASSED_ARB);
glGetQueryObjectivARB(query[0],
    GL_QUERY_RESULT_ARB, &nrDrawnPixels);
// nrDrawnPixels pixels were drawn
```

After n_{bins} rendering passes in which a new plane was drawn and an occlusion query was started, one retrieves the cumulative histogram of absolute frequencies H^{abs}. Finally, H^{abs} must be transformed into the histogram of relative frequencies h^{rel}. This is done on the CPU using (5) and (6).

3 Results

In order to evaluate the performance of our algorithm, we use a dual XEON 2.8 GHz workstation with 2GB of memory and a GeForce 8800GTX PCIe graphics cards with 768 MB. The GPU programming is done entirely in OpenGL and the Cg. We use Framebuffer Objects to render to texture.

We compare our method with NVIDIA's method [Green 2005] and a CPU method that calculates the histogram entirely on the CPU (the CPU implementation is of course compiled with compiler optimizations switched on). The comparison depends on two parameters: The first is the size of the image of which the histogram is calculated and the second is the number of bins in the histogram.

image size	our method	NVIDIA's method	CPU method
1024×1024	6.0 ms	70.7 ms	8.3 ms
512×512	1.9 ms	16.9 ms	2.1 ms
256×256	1.5 ms	6.1 ms	0.5 ms
128×128	1.4 ms	4.5 ms	0.13 ms
64×64	1.4 ms	4.4 ms	0.04 ms

Table 1: Comparison of our method, NVIDIA's method and the CPU method for the histogram computation of the X-ray image in Figure 4. The size of the image changes from 1024×1024 to 64×64 pixel. The number of histogram bins is 256. The measured times are averaged over 5000 iterations.

image size	our method	NVIDIA's method	CPU method
1024×1024	8.0 ms	72.5 ms	8.4 ms
512×512	2.5 ms	17.5 ms	2.1 ms
256×256	1.5 ms	6.1 ms	0.5 ms
128×128	1.4 ms	4.5 ms	0.15 ms
64×64	1.5 ms	4.4 ms	0.04 ms

Table 2: Comparison of our method, NVIDIA's method and the CPU method for the histogram computation of the teapot image in Figure 5. The size of the image changes from 1024×1024 to 64×64 pixel. The number of histogram bins is 256. The measured times are averaged over 5000 iterations.

We use the X-ray image in Figure 4 and the teapot image in Figure 5 for our evaluation. The histograms of these images are shown in Figure 6 and in Figure 7.

In the first experiment we varied the size of the image of which the histogram was computed. The images have a square size of 1024×1024, 512×512, 256×256, 128×128 and 64×64 pixels. The number of bins in the histogram was set to 256.

We see in Table 1 and Table 2 that by using images with 1024×1024 pixels our algorithm was about ten times faster than the algorithm that implemented NVIDIA's method. If the histograms were calculated for smaller images, our method was at least three times faster than NVIDIA's method. For large images with 1024×1024 pixels, our method was even faster than the CPU method. That means that our approach benefits from the vast filling rates of today's graphic cards. Only if small images were used, the CPU method was the fastest one of all three methods because of the setup costs for the GPU methods.

Although our method was 1 ms slower than the CPU method for small images, by using our GPU algorithm one avoids transfer costs between the CPU and the GPU and, additionally, the CPU remains idle for other computational work.

In practical approaches the computation time for the CPU method would be significantly higher as one has to add the stated transfer costs and has to convert the float data from the GPU to integer values or calculate the histogram on float images what is rather expensive.

In the second experiment we varied the number of bins in the histogram. The size of the image was set to 1024×1024 pixels. We see in Table 3 and in Table 4 that our method was for every number of bins faster than NVIDIA's method. For a large number of bins in the histogram, our method was about ten times faster than NVIDIA's method. For a small number of bins in the histogram, our method was at least four times faster than NVIDIA's method.

histogram bins	our method	NVIDIA's method	CPU method
256	6.0 ms	70.7 ms	8.3 ms
128	3.4 ms	36.0 ms	8.3 ms
64	2.1 ms	18.7 ms	8.4 ms
32	1.5 ms	10.0 ms	8.3 ms
16	1.1 ms	5.7 ms	8.3 ms

Table 3: Comparison of our method, NVIDIA's method and the CPU method for the histogram computation of the X-ray image in Figure 4. The number of bins in the histogram varies from 256 to 16. The size of the image is 1024×1204. The measured times are averaged over 5000 iterations.

histogram bins	our method	NVIDIA's method	CPU method
256	8.0 ms	72.5 ms	8.4 ms
128	4.4 ms	37.1 ms	8.4 ms
64	2.7 ms	19.2 ms	8.3 ms
32	1.8 ms	10.3 ms	8.3 ms
16	1.3 ms	5.8 ms	8.3 ms

Table 4: Comparison of our method, NVIDIA's method and the CPU method for the histogram computation of the teapot image in Figure 5. The number of bins in the histogram varies from 256 to 16. The size of the image is 1024×1204. The measured times are averaged values over 5000 iterations.

The CPU method was of course not affected by the number of bins in the histogram. If the histogram was calculated for large images with 1024×1024 pixels, our method was faster than the CPU method for every number of bins in the histogram. In this scenario, our method was at most eight times faster than the CPU method if the histogram had only a small number of bins. In these situations it would be reasonable to compute the histogram on the GPU, especially if the rest of the image processing algorithm is also computed on the GPU. The runtime of NVIDIA's method is strongly affected by the number of bins in the histogram. For a small number of bins in the histogram, it does better than the CPU. This aligns with a statement in [Green 2005] that NVIDIA's method is only feasible for a small number of bins.

To sum up our experiments, we can conclude that our method is up to ten times faster than NVIDIA's method. In every scenario our method was the fastest of both methods. For histograms with a small number of bins or for large images, our method was also faster than the CPU method.

The decision which method should be used depends on the application. If the image one has to calculate a histogram for is already available on the graphics card it is recommendable to calculate the histogram directly on the graphics card. If the image is only available in the computer's main memory one should not try to upload the image to GPU first and then calculate the histogram using our method.

4 Conclusions and Future Work

In this paper we presented a new approach to compute histograms on the GPU. Because several image processing algorithms that are based on histograms were implemented on the GPU, it is also required to compute histograms on the GPU. Because of architectural limitations of the GPU, it is not possible to compute histograms on the GPU in the same way as on the CPU. Therefore, new techniques

Figure 4: The X-ray image which was used for the results in Table 1 and Table 3.

Figure 5: The teapot image which was also used for the results in Table 2 and Table 4.

were proposed that compute histograms on the GPU in a different manner. However, these techniques were not fast enough.

Our method uses a fragment shader, occlusion queries and the depth-buffer to compute histograms efficiently on the GPU. The gray level values of the image for which the histogram will be calculated are taken as depth values. The depth-buffer in combination with occlusion queries can efficiently be used to count the number of gray values that fall into each histogram bin.

We showed in our experiments that our method is faster than an existing technique for the GPU-based computation of histograms. For a small number of bins in the histogram or for large images, our method is even significantly faster than a CPU implementation. If the computation of the histogram can be done on the GPU, one avoids transfer costs between the CPU and the GPU and, additionally, the CPU remains idle for other computational work.

In the future we will integrate our method for computing a histogram in several GPU-based image processing algorithms. Additionally, we will explore the possibilities for computing joint histograms on the GPU.

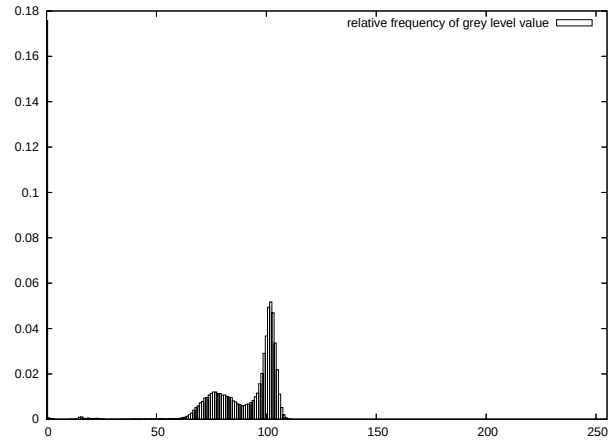

Figure 6: The histogram of relative frequencies for the X-ray image in Figure 4.

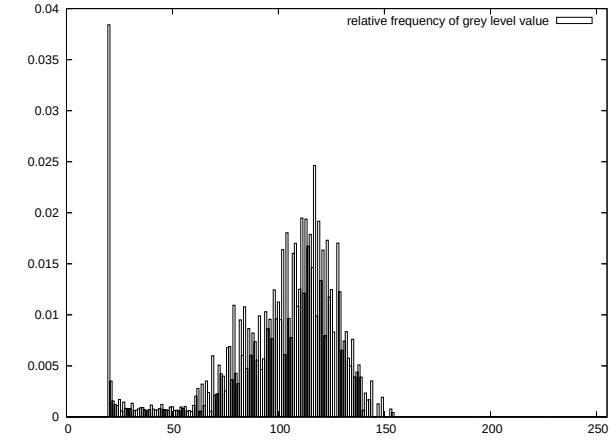

Figure 7: The histogram of relative frequencies for the teapot image in Figure 5.

References

COLLIGNON, A., MAES, F., DELAERE, D., VANDERMEULEN, D., SUETENS, P., AND MARCHAL, G. 1995. Automated Multimodality Image Registration using Information Theory. In *Information Processing in Medical Imaging, Proc.*, Y. Bizais, C. Barillot, and R. D. Paola, Eds. Kluwer Academic, KA, Dordrecht, 263–274.

DEUERLING-ZHENG, Y., LELL, M., GALANT, A., AND HORNEGGER, J. 2006. Motion Compensation in Digital Subtraction Angiography Using Graphics Hardware. *Comput Med Imaging Graph. 5*, 279–289.

FLUCK, O., AHARON, S., CREMERS, D., AND ROUSSON, M. 2006. Gpu histogram computation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, ACM Press, New York, NY, USA, ACM, 53.

FUNG, J., AND MANN, S. 2004. Computer Vision Signal Processing on Graphics Processing Units. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, IEEE, 93–96.

FUNG, J., AND MANN, S. 2005. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, ACM Press, New York, NY, USA, ACM, 849–852.

GREEN, S. 2005. Image Processing Tricks in OpenGL. In *Game Developers Conference*. http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_Image_Processing_Tricks.pdf.

HILL, D. L., STUDHOLME, C., AND HAWKES, D. J. 1994. Voxel Similarity Measures for Automated Image Registration. In *Visualization in Biomedical Computing 1994*, SPIE Proceedings, SPIE, R. A. Robb, Ed., vol. 2359, The International Society for Optical Engeneering, SPIE, 205–216.

KHAMENE, A. 2005. Automatic Registration of Portal Images and Volumetric CT for Patient Positioning in Radiation Therapy. *Medical Image Analysis 10*, 96–112.

KÖHN, A., DREXL, J., RITTER, F., KÖNIG, M., AND PEITGEN, H. O. 2006. GPU Accelerated Image Registration in Two and Three Dimensions. In *Bildverarbeitung für die Medizin 2006*, Springer, Berlin, 261–265.

LAROSE, D. A. 2001. *Iterative X-Ray/CT Registration Using Accelerated Volume Rendering*. PhD thesis, Carnegie Mellon University.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2005. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, 21–51.

PENNEY, G. P. 1999. *Registration of Tomographic Images to X-Ray Projections for Use in Image Guided Interventions*. PhD thesis, King's College, London.

RUMPF, M., AND STRZODKA, R. 2001. Level Set Segmentation in Graphics Hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, vol. 3, 1103–1106.

STRZODKA, R., DROSKE, M., AND RUMPF, M. 2004. Image Registration by a Regularized Gradient Flow - A Streaming Implementation in DX9 Graphics Hardware. *Computing 73*, 4, 373–389.

VIOLA, I., KANITSAR, A., AND GRÖLLER, M. E. 2003. Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages. In *Proceedings of IEEE Visualization 2003*, IEEE, K. M. G. Turk, J. van Wijk, Ed., 309–316.

WEIN, W. 2003. *Intensity Based Rigid 2D-3D Registration Algorithms for Radiation Therapy*. Master's thesis, Technische Universität München.

YANG, R., AND POLLEFEYS, M. 2005. A Versatile Stereo Implementation on Commodity Graphics Hardware. *Real-Time Imaging 11*, 1, 7–18.