A Self-Optimizing Histogram Algorithm for Graphics Card Accelerated Image Registration

Tom Brosch[1] and Roger Tam[2]

August 6, 2009

[1]MS/MRI Research Group, Faculty of Computer Science, University of Magdeburg
[2]MS/MRI Research Group, Department of Radiology, University of British Columbia

Abstract

This paper presents a new method to speed up image registration using graphics cards. In recent years, using graphics cards to implement image registration algorithms has become a reasonable way to reduce the computation time. While transforming images on the graphics card is rather simple, the computation of histograms as part of mutual information computation remains challenging. The performance of most algorithms depends greatly on the gray value distribution and is optimized for a small range of images only. In this paper a self-optimizing algorithm for histogram computation on the graphics card is presented which uses distribution information of the input images gained during a previous calculation step in order to adapt the histogram bin sizes to maximize the use of fast but limited local memory and avoid access collisions during subsequent calculations. The proposed approach has been evaluated using PD, T1 and T2 weighted MRI images of the brain revealing a four times speed-up on average compared to other graphics card implementations.

Contents

1 Introduction **2**

2 Method **3**

2.1 Histograms on the GPU . . . 3
2.2 Histograms with Adaptive Bin Size . . . 4
2.3 Algorithm Details . . . 4
2.4 Calculating Mutual Information . . . 7

3 Results **7**

4 Summary **8**

A Appendix **9**

1 Introduction

Image registration is an important preprocessing step in medical image analysis. The task of image registration is to find a transformation T which applied to the floating image v, aligns the floating image with the base image u. Viola and Wells formulated this problem as an optimization problem [6] defined as

$$\hat{T} = \arg\max_{T} I(u, T(v)) \tag{1}$$

with I the *mutual information* (MI) between the base image and the transformed floating image. There are several ways to compute mutual information[3]. One definition is given by

$$I(A,B) = \sum_{a}\sum_{b} p_{A,B}(a,b) \log \frac{p_{A,B}(a,b)}{p_A(a)p_B(b)} \tag{2}$$

where $p_{A,B}$ is the joint probability density function (PDF) and p_A and p_B are the marginal PDFs. A good way to estimate PDFs are histograms. Joint histograms are used to estimate joint PDFs. After calculating the joint histogram, the histograms of image A and B can be computed by summing all bins along the rows and columns of the joint histogram. Hence, it is sufficient to calculate the joint histogram only in order to compute mutual information.

The optimization problem is solved by iteratively transforming the floating image and calculating mutual information. Depending on the size and dimension of the images used and the number of degrees of freedom of the transformation, registration can take several minutes up to several hours. Since image registration is a time-consuming task it makes great demands on the underlying hardware.

In the recent decade, graphics cards have advanced to powerful processing units and have become well accepted in the field of scientific computations – especially for complex simulations. This reduces the hardware costs by reducing the need for large computer clusters, but the development of GPU programs has still been labor-intensive. With the introduction of the *Compute Unified Device Architecture* (CUDA), NVIDIA made it easy to use the computational power of today's graphics cards for a large field of applications while reducing the development costs. So GPU implementations have found their way into medical image processing applications like image registration. Popular examples come from Shams, who presented in [4] a method for image registration on the GPU, which is 25 times faster than image registration on a single same-generation CPU. Other examples come from Chu [1]. He presented a method for non-rigid registration of breast MRI images using an adapted version of Shams' method for calculating histograms. This way, a 40 times speed-up compared to the original CPU implementation has been reached.

With CUDA, NVIDIA provides a technology to perform general purpose calculations on their graphics cards. The focus is to support the user in using this technique for highly parallel computing. The higher the number of independent operations that can be processed in parallel the higher the performance. There are also different memory units on a graphics card. The type of memory requiring the most consideration for optimization is the shared memory, which is the fastest memory but typically limited to only 16 KB. The remaining graphics card memory, that is usually refered to as global memory, is much larger (256 MB and more) but slower. Detailed information about CUDA can be found in the "Programing Guide" [2].

In this paper, a self-optimizing algorithm for computing a series of histograms, such as those used in image registration, is presented. The algorithm uses the knowledge of previous calculations to speed up the computation of following histograms by using the intensity distributions of the input images to adapt the histogram bin sizes in order to maximize the use of shared memory and avoid access collisions. This way, joint histograms can be computed up to five times faster on the GPU than other current GPU algorithms. In the next Section, we give a brief explanation of Shams' method [5], which is a recently proposed approach

for calculating histograms on the GPU. In Section 2 we present our approach, followed by a validation of our method and results in Section 3. The last section summarizes our conclusions.

2 Method

2.1 Histograms on the GPU

The calculation of histograms without parallelization is simple. The first step is to initialize all bins of the histogram to zero. Then for each pixel of the image the corresponding bin counter is incremented. A principal algorithm for the GPU is very similar. Instead of looping through the image, one thread per pixel is started which reads the pixel value and increments the corresponding bin counter. If two threads read the same intensity value they will increment the same bin counter so these operations need to be synchronized.

There are two principal strategies to overcome this issue. The most obvious solution is to let each thread calculate its own partial histogram. This method was first presented in [5]. After the calculation of partial histograms is finished all partial histograms need to be summed up to get the final histogram. In order to minimize slow global memory access all partial histograms should be placed in shared memory. The number of bits b that can be used for each bin can be calculated with Equation 3 ([5]).

$$b = \frac{s_{\max} \times 32}{B \times N_b} \tag{3}$$

where $s_{\max}$ is the number double words that fit into shared memory, B is the number of bins of the histogram and N_b is the number of threads.

Most of todays graphics cards have a shared memory of 16 KB which would fit 4096 double words. To maximize computational power the program should not have less than 128 threads, which corresponds to the number of calculation units typically available. For accurate MI computation with most medical image data, such as MRI, 128 bins per histogram would be considered a minimum, which would result in $128^2 = 16384$ bins for the joint histogram. Thus, there is less than one bit available for each bin so the fast shared memory cannot be used in a straightforward manner.

In order to maximize the use of limited memory, Shams presented a second method [5] where several threads share one bin. This involves the need for synchronizing thread access of conjointly used memory, which is not possible in general without synchronization primitives. However, since all 32 threads of a collective thread unit, called a thread warp, perform the same operation at any time, it is possible to simulate a mutex for groups of 32 threads. Every time a thread tries to write to shared memory the counter will be tagged with the thread ID. After writing, the written value will be read back and by checking the thread tag, each thread can check whether the writing operation succeeded and repeat it if necessary. The storage of tags reduces the number of bits that can be used for the counter itself. To comply with the shared memory alignment requirements, every bin is stored in a double word. Shams pointed out that the algorithms performs best with eight thread warps each calculating their own partial histogram. With a shared memory of 16 KB and eight histograms only 512 bins would fit into shared memory compared to 128^2, which are needed for accurate MI computation. To overcome this problem, only a specific bin range is calculated and the algorithm calculates each bin range iteratively.

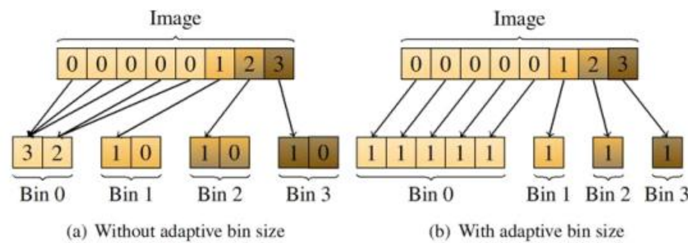

Figure 1: Comparison of histogram configurations with and without adaptive bin size. Histograms with adaptive bin size use the available memory more efficient resulting in less thread collisions and less overflows.

2.2 Histograms with Adaptive Bin Size

Usually, the distribution of gray values is non-uniform. That means that some bins of the histogram need to be incremented more frequently than others resulting in many overflows and global updates for bins that are frequently used and many empty or nearly empty bins. Our algorithm adapts the histogram bin sizes to fit the image data. Bins that are more likely updated are allocated more bytes than others. Bins that are nearly never used are not cached in shared memory at all saving space for other bins.

This idea is illustrated in Fig. 1. In the example, a histogram is calculated with eight threads for an image with eight pixels. The first case shows a histogram with equally sized bins. Each bin is allocated two bytes, and each byte is filled independently. Because there are five pixels with the gray value zero, the threads collide while updating bin zero. On the other hand the bins one, two and three each occupy one byte that is not used. A histogram with adaptive bin size is able to balance bin updates much better avoiding collisions and unused memory completely.

2.3 Algorithm Details

The algorithm can be divided into two major parts – the calculation of an optimal histogram configuration and the calculation of histograms itself. The histogram configuration contains for each bin the number of bytes that are used to cache it in shared memory. Since the calculation of an optimal histogram configuration needs an estimate of the probability density function the first histogram calculation is not optimized. The optimization of the histogram configuration is time consuming, but since the histograms typically do not change very much during the registration process, the histogram configuration needs to be calculated only once at the beginning. So when many histograms are computed in succession the costs for the first calculation and histogram optimization amortizes well over the complete registration process. To facilitate an explanation of the optimization process, we first describe how we calculate histograms with adaptive bin sizes.

Calculate Histograms An overview of the algorithm is given in Algorithm 1. The first step is to initialize the histogram in global and shared memory. After initializing the memory the histogram is updated for each pixel of the image, with the number of available threads determining the number of pixels that can be processed simultaneously. Each bin has a *logical bin position*, which is determined by the gray value of the current pixel and the number of gray values which fall into the same bin. Then, the *bin parameters* for the current bin are read. These parameters specify whether a bin is cached in shared memory and if so how

many bytes are used to store it as well as the position in shared memory. In case a bin is spread over more than one byte each thread group increments a separate byte which is identified by the bin parameters and the current thread ID.

Before incrementing the bin it is checked for a potential overflow. The maximum value of the bin depends on the number of threads that share a bin, because some bits of the bin are used for synchronization. The more threads that share one byte the more bits are used for synchronization, reducing the maximum value of the bin. If an overflow needs to be handled each thread writes zero to the bin and the incremented value is updated in global memory, otherwise the bin is incremented in shared memory. A unique thread ID is also written in the upper bit range. If two threads try to write to the same memory location only one thread will succeed. After reading the written value the thread ID can be checked to test if the last writing operation succeeded. If the write operation does not succeed the procedure is repeated.

Algorithm 1 Calculation of histograms with adaptive bin size

```
 1: initialize all bins of the histogram to zero
 2: for all p, p ∈ image do {This loop is processed in parallel}
 3:   get logical bin position b_l of p and the corresponding bin parameters
 4:   if bin is cached then
 5:     get physical bin position b_p depending on the bin parameters and thread ID
 6:     repeat
 7:       if bin b_p is full then
 8:         try to set bin b_p to zero
 9:         if last writing operation succeeded then
10:           update global memory
11:         end if
12:       else
13:         try to increment bin b_p
14:       end if
15:     until last writing operation succeeded
16:   else {bin is not cached}
17:     update global memory
18:   end if
19: end for
20: write cached values in shared memory to global memory
```

Histogram Configuration When incrementing a bin, each bin creates costs. The greatest costs are caused by collisions in shared or global memory and by global memory updates. The aim of the optimization is to minimize the expected costs. To achieve this, a bin level l_i is calculated for each bin i. The bin level indicates how many bytes are used for caching a bin in shared memory. A level of 0 indicates that a bin is not cached at all otherwise the used bytes are given by $b_i = 2^{l_i - 1}$. The expected costs are defined as

$$\text{Costs} = \sum_{i=0}^{n_b - 1} c_{gu,i} + c_{sc,i} + c_{gc,i} \quad (4)$$

with n_b number of bins, $c_{gu,i}$ costs caused by global updates of bin i, $c_{sc,i}$ costs caused by collisions in shared memory and $c_{gc,i}$ costs caused by collisions in global memory by bin i. The total costs of global memory updates are estimated as the costs for one update c_{gu} times the expected number of global updates and given

by

$$c_{\text{gu},i} = \begin{cases} c_{\text{gu}} \frac{h_i}{b_{\max,i}} & \text{if } l_i > 0 \\ c_{\text{gu}} h_i & \text{if } l_i = 0 \end{cases} \tag{5}$$

with h_i the value of bin i and $b_{\max,i}$ the number of different values that can be stored in bin i. The maximum number that can be stored in bin i depends on the number of bits that are used to store the bin counter. With a bin level of 1 32 threads share one bin so $\log_2(32) = 5$ bits are used to store the thread tag leaving only 3 bits to store the bin counter. Hence, $2^3 = 2^1 \times 2^2$ differnt values can be stored in a bin with a bin level 1. Increasing the bin level by 1 doubles the number of bytes that are used to cache a bin and halves the number of threads that share one bin reducing the number of bits used to store the thread tag by 1. Thus, one more bit can be used to store the bin counter doubling the number of different values of the bin. From this it follows that $b_{\max,i} = 2^{l_i} \times 2^2 = 2^{l_i+2}$.

Next, we calculate the expected number of collisions in shared memory. Consider we are given n threads which try to access the same bin i each with a probability of p. Then the probability $u_{k,i}$ that exactly k threads try to update bin i is given by

$$u_{k,i} = \mathrm{B}(k \mid p,n) = \binom{n}{k} p^k (1-p)^{n-k} \tag{6}$$

The expected number of updates of bin i is

$$\mathrm{E}[u_i] = \sum_{i=0}^{n} i\mathrm{B}(i \mid p,n) = np \tag{7}$$

This yields to the expected number of collisions given by

$$\mathrm{E}[c_i] = \sum_{i=1}^{n} (i-1)\mathrm{B}(i \mid p,n) = np + (1-p)^n - 1 \tag{8}$$

The number of threads n which try to access the same bin depends on the bin level. With a bin level of 1 $32 = 2^5 = 2^{6-1}$ threads share one bin. Increasing the bin level by 1 halves the number of threads that share one bin, so $n = 2^{6-l_i}$. The expected costs of bin i with respect to its bin level l_i is given by

$$c_{\text{sc},i} = \begin{cases} c_{\text{sc}} h_{\text{sum}} \frac{np+(1-p)^n-1}{n} & \text{if } l_i > 0 \\ 0 & \text{if } l_i = 0 \end{cases} \tag{9}$$

with $h_{\text{sum}} = \sum_i h_i$ and $p = h_i / h_{\text{sum}}$.

Finally, we need the expected costs caused by global memory collisions. Analogous to the costs caused by collisions in shared memory these costs are given by

$$c_{\text{gc},i} = \begin{cases} 0 & \text{if } l_i > 0 \\ c_{\text{gc}} h_{\text{sum}} \frac{np+(1-p)^n-1}{n} & \text{if } l_i = 0 \end{cases} \tag{10}$$

with $n = n_{\text{T}}$, the total number of threads, and $p = h_i / h_{\text{sum}}$. The costs caused by collisions in global memory due to overflows in shared memory are negligible and therefore ignored.

In order to find an optimal histogram configuration, each bin level is set to zero and the expected costs of each bin are calculated. The number of available bytes are set to the number of bytes used to store histograms

in shared memory. Then the *relative benefit* defined as the expected decrease of costs divided by the bytes that are needed to increase the bin level are computed for each bin. As long as there are bytes available, the bin level of the bin which promises the greatest relative benefit is incremented and the number of available bytes is decremented by the number of bytes used for the last optimization step. An optimal histogram configuration is found when there are no more bytes available or all bins have reached the maximum bin level 6 which means that no bins are shared by more than one thread.

2.4 Calculating Mutual Information

As outlined in the introduction, the computation of mutual information requires an estimate of the PDFs of both images and the joint PDF. Histograms and joint histograms can be used to provide such an estimate. In the previous section, we have shown a method to calculate histograms on the graphics card. Since the calculation of joint histograms is separable in each dimension, this method can be used to calculate joint histograms as well. Therefore, the 2D bin index of the joint histogram (x, y) is mapped to the 1D index i by $i = w \times y + x$ where w is the width of the joint histogram. The PDFs themselves are given by dividing each bin counter by the number of voxels of one image. Then, mutual information can be computed using Equation 2.

3 Results

In this section, we present the results of two experiments, one to compare the overall registration speed between our new algorithm and Shams' methods, and the other to analyze how much of the runtime of each registration iteration is occupied by histogram calculation. For both tests, we used a data set consisting of PD, T1 and T2 weighted MRI images of the brain of two patients with a resolution of $256 \times 256 \times 60$ voxels. All image pairs were formed using all combinations of images of the same patient. Between 300 and 400 iterations were necessary in order to register two images. Details about the graphics card that were used to perform all tests are summarized in Appendix A, Table 1.

During the first test, joint histograms are calculated for different numbers of bins for all image pairs of the test set using our approach and the two methods presented by Shams. Then, all results for the same number of bins and the same method have been averaged to get the mean runtime depending on the number of bins and algorithm. Since the first calculation of the method with adaptive bin size takes a lot more time (up to three seconds) than all following calculations, the joint histogram is calculated twice and only the runtime of the second calculation was taken into account. The results are summarized in Fig 2. The runtime of Shams' first and second method increases with increasing number of bins, while the runtime of our approach is nearly constant. With Shams' first method, only a fixed number of bin can be calculated at a time. So with more bins more iterations are needed to calculate the complete histogram resulting in a longer runtime. With the second method, all bins are filled during one iteration, but the amount of fast shared memory that can be used for each bin decreases with increasing number of bins, so the overall runtime increases too.

Our proposed method performed best during this test. Because only one iteration is needed to calculate all bins, it is faster than Shams' first method. In addition, the shared memory is used much more efficiently than with Shams' second method resulting in a shorter runtime when many bins are calculated. The more complicated calculation due to the interpretation of the bin configuration introduces some algorithmic overhead, which could have led to a worse runtime. Our tests have proven that this overhead is small in comparison to the benefit of an optimal histogram configuration so the algorithm is faster even with a very small number of bins. However, the difference becomes much more clear as more bins are used.

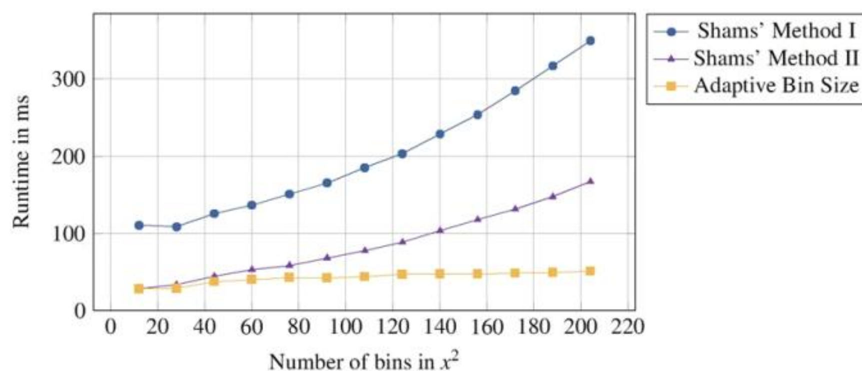

Figure 2: Comparison of the time need to compute mutual information using different histogram algorithms for increasing number of bins.

In the second test, we measured the proportion of the total runtime of the registration process required for histogram computation, and compared it to that required by transformations and computation of mutual information. Therefore, we registered each image pair of our data set using different histogram algorithms. Each joint histogram was calculated using 240×240 bins. The optimization algorithm used to maximize mutual information was a combination of Powell's method which uses Brent's method to perform the line optimization step. During each optimization step, the floating image needs to be transformed according to the new transformation parameters. Then the joint histogram of the base image and the transformed floating image was computed as a part of the mutual information computation. For each step, three intermediate times were recorded: 1) the time needed to calculate the joint histogram (this includes the time needed to calculate the optimal histogram configuration in case of our method), 2) the time needed to calculate mutual information including the time needed to calculate the joint histogram and 3) the two simple histograms and the time needed for one complete optimization step including the time needed to transform the floating image and to calculate mutual information. Then, for each method, the averaged time over all iterations and all images to perform each of these steps has been calculated. The results are illustrated in Figure 3.

The first observation is that for all algorithms the total runtime is dominated by the time needed to compute joint histograms. The time needed to compute MI excluding the time needed to calculate joint histograms and the time needed to transform the images are nearly negligible when performing these operations on the graphics card. In Section 2.3, we claimed that the costs for the first calculation and histogram optimization amortizes well over the complete registration process. Figure 3 supports this claim because, even though the calculation of the first histogram and the histogram calculation is included in the averaged histogram time, it is much faster compared to Shams' methods. Simply by using our approach to calculate joint histograms, image registration can be performed three to four times faster compared to other graphics card implementations.

4 Summary

Other publications have shown that GPU implementations can greatly enhance the performance of image registration compared to CPU implementations due to the vast computational power of today's graphics cards. While the calculation of transformations is rather simple on the GPU, counting and gathering al-

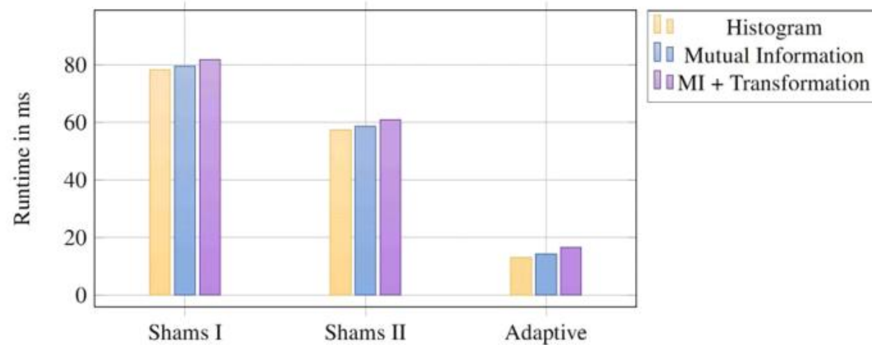

Figure 3: Averaged time to compute histograms, mutual information and one optimization iteration including the computation of mutual information and transforming the image for different histogram algorithms.

gorithms like calculating histograms are challenging due to the lack of synchronization primitives. There are some good solutions that depend greatly on the very limited amount of fast memory, and we have used this previous work as a starting point for designing our new algorithm. As with Sham's methods, our algorithm also computes histograms and joint histograms on the graphics card. The main difference is the way the shared memory is used. Shams used the shared memory to cache all bins of the histogram, no matter whether they are updated frequently or not. Our new approach tries to overcome this issue by calculating a histogram configuration based on a previously calculated histogram. Depending on the parameter difference between different optimization steps, the histograms are very similar. In case of large initial misregistation, a new histogram configuration calculation might be necessary after moving a specific distance in order to have optimal caching even at the end of the registration process. However, our evaluations have shown that for common registration problems such a resetting step is not necessary.

The difference in performance between CPU and GPU implementation is dominated by the differences in the computational power of CPUs and GPUs so such a comparison might not be appropriate to judge the quality of a GPU implementation itself. Therefore, we compared our implementation with other GPU implementations in order to provide informative results. We have shown, that it is possible to speed up common GPU implementations by improving the memory usage. It is possible to calculate an optimal histogram configuration, if a good estimate of the distribution is available. In common registration tasks, the calculation of many similar histograms is required, therefore a previous histogram can provide a good estimate of optimal memory usage for subsequent histogram calculations. Furthermore, we showed that calculating histograms is the bottleneck of image registration using mutual information. Even though the calculation of an optimal histogram configuration is very slow compared to the histogram calculation, the overall runtime could be reduced significantly by reducing the computation time of histograms. So implementing the calculation of an optimal histogram configuration on the GPU will further the reduce the runtime in the future.

A Appendix

Table 1: Graphics Card Specification

Model	NVIDIA GeForce 9600 GT
Processor Cores	64
Graphics Clock	650 MHz
Processor Clock	1625 MHz
Graphics Memory	1024 MB
Shared Memory	16 KB
Max number of threads per block	512
Warp Size	32

References

[1] Mei Yi Chu. *Mutual Information based Non-Rigid Image Registration using Adaptive Grid Generation: GPU.* PhD thesis, University of Texas at Arlington, 2008. 1

[2] C. NVIDIA. Programming Guide 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, 2008. 1

[3] J. P. W. Pluim, J. B. A. Maintz, and M. A. Viergever. Mutual-information-based registration of medical images: a survey. *IEEE transactions on medical imaging*, 22(8):986–1004, 2003. 1

[4] R. Shams and N. Barnes. Speeding up mutual information computation using NVIDIA CUDA hardware. In *Proc. Digital Image Computing: Techniques and Applications (DICTA)*, pages 555–560, Adelaide, Australia, December 2007. 1

[5] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, Gold Coast, Australia, December 2007. 1, 2.1, 2.1

[6] Paul Viola and William M. Wells III. Alignment by Maximization of Mutual Information. *International Journal of Computer Vision*, 24(2):137–154, 1997. 1