3D Registration based on Normalized Mutual Information
Performance of CPU vs. GPU Implementation

Florian Jung, Stefan Wesarg

Interactive Graphics Systems Group (GRIS), TU Darmstadt, Germany
stefan.wesarg@gris.tu-darmstadt.de

Abstract. Medical image registration is time-consuming but can be sped up employing parallel processing on the GPU. Normalized mutual information (NMI) is a well performing similarity measure for performing multi-modal registration. We present CUDA based solutions for computing NMI on the GPU and compare the results obtained by rigidly registering multi-modal data sets with a CPU based implementation. Our tests with RIRE data sets show a speed-up of factor 5 to 7 for our best GPU implementation.

1 Introduction

The registration of medical image data plays an increasing role in diagnosis as well as image-guided interventions. A widely used similarity measure for multi-modal registration is the normalized mutual information (NMI) metric [1]. In this work, we investigate to which extent the execution of an NMI based rigid registration can be sped up if parts of the algorithm are executed in parallel on graphics cards. A CUDA-based implementation on the GPU is compared to a CPU based solution for 3D rigid registration of medical image data.

By studying recent publications, attempts for employing GPU hardware and partially CUDA for solving image registration problems can be found. Köhn et al. [2] investigated the performance gain for rigid registration using a sum of squared differences metric, that performs well for intra-modality registration but has limitations for inter-modality registration [3]. Programming was entirely done on the GPU employing GLSL. An approach for 2D/3D registration based on automatic differentiation has been introduced by Grabner et al. [4]. It uses Cg for the GPU part of the algorithm, and a performance gain of factor 6 is reported. Muyan-Özcelik et al. [5] describe a CUDA implementation of Thirion's deformable registration algorithm [6] with a speed-up of 55 times compared to a CPU implementation. The problem of MI based registration using CUDA has been investigated by Shams et al. [7]. Their solution consists in an approximation of the probability density function, which is part of the MI based similarity measure calculation, employing a down-sampled joint histogram. In contrast to their work, we compute NMI based on histograms with a much higher number of bins for assuring a registration accuracy for the GPU approach that is equal to a CPU implementation.

2 Materials and Methods

The registration of two data sets requires their initial resampling in order to have isotropic voxels of equal resolution. This has to be done only once thus representing a negligible portion of the overall computation time. The other steps involved in our implementation are the interpolation of voxel data, the computation of NMI as similarity measure, an optimization step, and the application of a transformation to one of the data sets. They are executed iteratively until the optimization does not further improve the value of the similarity measure. The optimization step is done using a downhill simplex algorithm taken from the *Numerical Recipes* [8]. An initial test of the CPU reference implementation focused on the execution time of the remaining three steps. Obviously, the greatest speed-up can be achieved if steps which are computationally demanding are shifted to the GPU. Figure 1 shows the result of such an examination for the registration of a CT and an MRI data set of the head. It can be seen that NMI computation consumes most of the time. Thus, we decided to develop a CUDA implementation for this step.

Employing an NMI metric requires the computation of two 1D histograms and one 2D joint histogram. For that, each voxel can be processed independently from its neighbors. In order to avoid that different threads concurrently increase the value of the same histogram bin, this requires mutual exclusion, a technique that is realized in CUDA with the AtomicAdd() function.

A straight-forward solution would be a direct port of the CPU implementation to CUDA. There, the number of threads would be equal to the number of voxels. It is obvious, that this will not result in fast code, since in case of 1D histograms with 256 bins which are stored in global memory only a maximum of 256 threads can write simultaneously, and this only if each of them accesses a different bin – which is rather unlikely in reality. In the following, we present two solutions with an increasing efficiency for the computation of 1D and 2D histograms as well as the calculation of the NMI metric. In our case, the 1D histograms have 256 bins and the 2D histogram is made up of 256×256 bins.

2.1 Independent Computation of 1D and 2D Histograms

We initialize each block with 256 threads – each of them computing the corresponding bins for one voxel position in both data sets. Each block holds

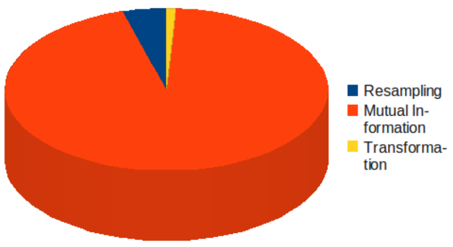


Fig. 1. The relative execution times for the steps being candidates for a CUDA implementation. It can be seen that NMI computation takes most of the time during a single iteration.

two 1D histogram copies in its corresponding shared memory, thus we benefit from the fast access to this type of memory. After initialization and filling of the 1D histograms in each block, each thread writes back the value of one bin from shared memory to the two resulting 1D histograms in global memory using AtomicAdd().

Due to its limited size, it is not possible to store also the 2D histogram in shared memory. Therefore, we hold 256 additional copies of the 2D histogram in global memory, and each thread per block writes into a different 2D histogram. This approach requires an additional but very small 'wrap-up' kernel which consists of 256 blocks each with 256 kernels. Thus, each thread accumulates all values of the same 2D bin over the 256 2D histograms and writes the sum to the final 2D histogram – without any read/write conflicts.

2.2 Computation of 1D Histograms from the 2D Histogram

In fact, there is no need for the computation of the two 1D histograms if the 2D histogram is available: a simple projection of the 2D bin values to the two axes provides the information for the 1D histograms. In this second implementation, we compute in a first kernel the 2D histogram as described above. Afterwards, a second kernel is launched, where each thread computes the sum over all bins of one row and one column of the 2D histogram, respectively. For an optimal load balancing this second kernel contains 16 blocks each with 16 threads.

2.3 Interpolation and NMI Computation

The CPU reference implementation uses a nearest neighbor interpolation scheme due to its faster execution compared to a trilinear interpolation. In the case of GPU usage we could get a trilinear interpolation virtually for free. If texture memory is used instead of global memory for storing the image data on the graphics card, a fetch to the 3D texture memory delivers the trilinearly interpolated value. However, for the sake of comparability between CPU and GPU implementation, we use nearest neighbor interpolation also in the CUDA code.

The computation of NMI can be done efficiently using a reduction scheme. We employ code from the CUDA SDK (http://www.nvidia.com/object/cuda) vector reduction example that has been adapted to compute first the probability density functions (see [1]) followed by the reduction itself.

3 Results

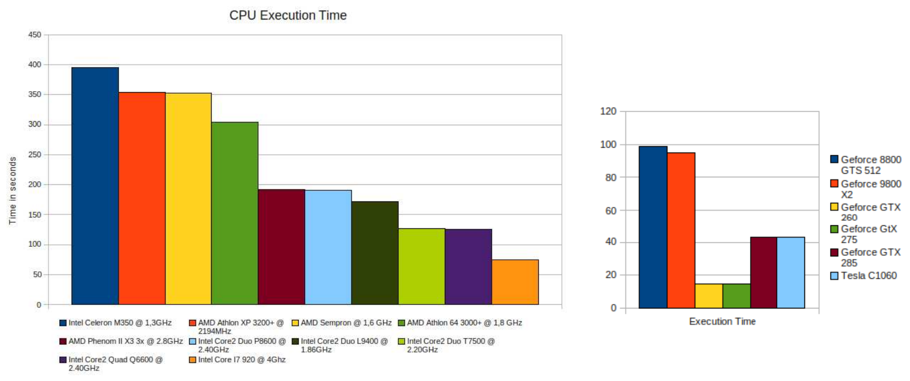
The tests of our implementation have been two-fold. Firstly, we used two 8 bit head data sets from our own data base – one CT and one MRI data set – of the same patient for testing the behavior over different hardware generations. The CT data set contained $512\times512\times49$ voxels, the MRI data set $256\times256\times50$ voxels. On the CPU (Fig. 2) an increasing speed which each new processor version could be noticed. It mainly scaled with the CPU's frequency, since we used a single-threaded implementation. Surprisingly, we could realize a lower performance of the CUDA implementation on graphics cards which are considered to be the fastest compared to the ones just below in the NVIDIA lineup (Fig. 2).

Secondly, we tested and compared the execution times on current hardware. CPU performance was measured on an Intel Quad Core Q6600 processor with 2.4 GHz (single-threaded code). As GPU an NVIDIA GeForce GTX 260 was chosen. For these tests, we used data sets from the open RIRE data base (http://www.insight-journal.org/rire). The results are shown in tab. 1. There, only the times for the most efficient (second) CUDA implementation are given revealing a speed-up of factor 5 to 7.

4 Discussion

We have presented new approaches for performing an NMI based 3D registration employing the GPU. With efficient CUDA implementations, a significant speed-up compared to a CPU implementation could be achieved. But, we compared our approach only to a single-threaded CPU solution. Employing multi-threaded programming on multi-core CPUs would reduce the performance gain of the GPU solution. Current limitation for a better performance on the GPU is the available hardware. Our best approach holds 256 copies of the 2D histogram in global memory due to the limited size of shared memory. If the latter one increases – as expected for one of the next generations of NVIDIA GPUs – each block could have its one 2D histogram for much faster read/write access.

Compared to an existing MI based registration approach [7], we use the full histograms – instead of a down-sampled version – for avoiding potential problems with registration accuracy. Here, our implementation is performing



(a) CPU (b) GPU

Fig. 2. The execution time for the registration of the CT and MRI head data set for different CPUs (a) as well as different GPUs (b). Note the different scaling of both diagrams!.

Table 1. The execution times needed for the registration of image data sets from the RIRE data base (http://www.insight-journal.org/rire/).

RIRE patient	Time CPU	Time GPU	Speed-up
001: CT – MR T1	46.64 s	8.99 s	5.2
001: CT – PD	86.80 s	14.34 s	6.0
001: MR T1 – PET	64.18 s	9.45 s	6.8
001: CT – PET	42.17 s	7.43 s	5.7
003: CT – MR T1	65.14 s	9.37 s	7.0
006: CT – PET	79.26 s	11.06 s	7.2
007: CT – MR T1	68.38 s	10.41 s	6.6
008: PET – PD	47.23 s	7.46 s	6.3

better – Shams et al. note that using an exact histogram makes their GPU solution slower than the CPU – which is, to be fair, also due to the fact that we can use the AtomicAdd() function, which had to be simulated in software in the aforementioned work.

Finally, we could perceive the strange behavior of the GTX 285 (240 cores, 1.4 GHz) compared to the GTX 260 (216 cores, 1.2 GHz) – being more than 3 times slower. Discussions with other people in the CUDA community indicated that the reason for that are different memory partitions for both GPUs. Future work will focus on taking this into account for getting the best out of the latest GPU generations.

References

1. Pluim JPW, et al. Mutual-information-based registration of medical images: a survey. IEEE Trans Med Imaging. 2003;22(8):986–1004.
2. Köhn A, et al. GPU accelerated image registration in two and three dimensions. In: Proc BVM; 2006. p. 261–5.
3. Hill DLG, et al. Medical image registration. Phys Med Biol. 2001;46(3):R1–R45.
4. Grabner M, et al. Automatic differentiation for GPU-accelerated 2D/3D registration. Lect Notes Computer Sci Eng. 2008;64:259–69.
5. Muyan-Özcelik P, et al. Fast deformable registration on the GPU: a CUDA implementation of demons. In: Proc ICCSA; 2008. p. 223–33.
6. Thirion JP. Image matching as a diffusion process: an analogy with Maxwell's demons. Med Image Anal. 1998;2(3):243–60.
7. Shams R, et al. Speeding up mutual information computation using NVIDIA CUDA hardware. In: Proc DICTA; 2007. p. 555–60.
8. Press WH, et al. Numerical recipes in C: the art of scientific computing. 2^{nd} ed. Cambridge University Press; 1992.