A Parallel GPU Algorithm for Mutual Information Based 3D Nonrigid Image Registration

Vaibhav Saxena[1], Jonathan Rohrer[2], and Leiguang Gong[3]

[1] IBM Research - India, New Delhi 110070, India
vaibhavsaxena@in.ibm.com
[2] IBM Research - Zurich, 8803 Rüschlikon, Switzerland
jon.rohrer@alumni.ethz.ch
[3] IBM T.J. Watson Research Center, NY 10598, USA
leiguang@us.ibm.com

Abstract. Many applications in biomedical image analysis require alignment or fusion of images acquired with different devices or at different times. Image registration geometrically aligns images allowing their fusion. Nonrigid techniques are usually required when the images contain anatomical structures of soft tissue. Nonrigid registration algorithms are very time consuming and can take hours for aligning a pair of 3D medical images on commodity workstation PCs. In this paper, we present parallel design and implementation of 3D non-rigid image registration for the Graphics Processing Units (GPUs). Existing GPU-based registration implementations are mainly limited to intra-modality registration problems. Our algorithm uses mutual information as the similarity metric and can process images of different modalities. The proposed design takes advantage of highly parallel and multi-threaded architecture of GPU containing large number of processing cores. The paper presents optimization techniques to effectively utilize high memory bandwidth provided by GPU using on-chip shared memory and co-operative memory update by multiple threads. Our results with optimized GPU implementation showed an average performance of 2.46 microseconds per voxel and achieved factor of 28 speedup over a CPU-based serial implementation. This improves the usability of nonrigid registration for some real world clinical applications and enables new ones, especially within intra-operative scenarios, where strict timing constraints apply.

1 Introduction

Image registration is a key computational tool in medical image analysis. It is the process of aligning two images usually acquired at different times with different imaging parameters or slightly different body positions, or using different imaging modalities, such as CT and MRI. Registration can compensate for subject motion and enables a reliable analysis of disease progression or treatment effectiveness over time. Fusion of pre- and intra-operative images can provide guidance to the surgeon. Rigid registration achieves alignment by scaling, rotation and translation. However, most parts of the human body are soft-tissue

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 223–234, 2010.

structures and more complex transformation models are required to align them. There is a variety of so-called nonrigid registration algorithms. A major obstacle to their widespread clinical use is the high computational cost. Nonrigid registration algorithms typically require many hours to register 3D images prohibiting interactive use [3]. Some implementations of nonrigid registration algorithms achieve runtimes in the order of minutes [11,5]. However, they typically run on large parallel systems of up to more than hundred processors. Acquisition and maintenance of such architectures is expensive and therefore the availability of such solutions is very limited.

In this paper, we present a CUDA-based implementation of a mutual information based nonrigid registration algorithm on the GPU, which achieves significant runtime acceleration and in many cases even sub-minute runtimes for multimodal registration of 3D images. Our solution provides low-cost fast nonrigid registration, which hopefully will facilitate a widespread clinical use. We present data partitioning for 3D images to effectively utilize large number of threads supported on the GPU, and optimization techniques to achieve high memory throughput from the GPU memory spaces. The proposed implementation achieves consistent speedup across different datasets of varying dimensions. The registration algorithm bases on a B-spline transformation model [15,13]. This approach has been used successfully for the registration of a variety of anatomical structures, such as the brain, the chest [7], the heart, the liver and the breast [13]. Mutual information is the most common metric for both monomodal and multimodal registration [10]. There are fast GPU implementations of nonrigid registration algorithms, but most of them are limited to monomodal registration [8,14,2]. The only multimodality-enabled implementation we are aware of is [17], which, however, uses only 2D textures and implemented with OpenGL and GLSL. Another similar work implemented on the Cell/B.E. is reported in [12]. However, the Cell/B.E. and GPU represent two very contrasting architectures and require different optimization approaches for achieving good performance.

The next section will provide an overview of the registration method. Section 3 will briefly describe the GPU architecture followed by section 4 of discussion of proposed parallel algorithm implementation. The experimental evaluation will be discussed in section 5 followed by conclusion in section 6.

2 Mutual Information Based Nonrigid Image Registration

Nonrigid image registration is the process to compute a nonlinear mapping or transformation between two images (2D or 3D). One of the images is called reference or fixed image and other one is called floating or moving image. We implemented a well-known mutual information based nonrigid image registration algorithm which models the transformation using B-splines. A set of control points are overlaid on the fixed image and transformation can be generated by letting these control points move freely. The transformation $T(\mathbf{x};\mu)$ is obtained by B-spline interpolation with transformation parameters μ that are B-spline

coefficients located at the control points. The degrees of freedom of control points are governed by the spacing between the points.

To register fixed image f_{fix} to moving image f_{mov}, an optimal set of transformation parameters μ are computed that best align the fixed and transformed moving image (i.e. establish similarity between them). To support the registration of images obtained from different modalities, we use negative of Mutual information as similarity metric. A gradient descent optimizer with feedback step adjustment [6] is used to iteratively obtain the optimal set of transformation parameters (coefficients) that minimize the similarity metric.

Mathematically, T provides a mapping of point in the fixed image space with coordinates $\mathbf{x}_{fix}$ to the point in the moving image space with coordinates $\mathbf{x}_{mov}$

$$\mathbf{x}_{mov} = T(\mathbf{x}_{fix}; \mu)$$

Using above mapping, we can obtain warped (transformed) moving image f'_{mov}

$$f'_{mov}(\mathbf{x}) = f_{mov}(T(\mathbf{x}_{fix}; \mu))$$

An optimal set of transformation parameters μ make the images f'_{mov} and f_{fix} comparable (similar) to each other based on a similarity metric.

The mutual information calculation is based on a Parzen estimation of the joint histogram $p(i_{fix}, i_{mov})$ of the fixed and the transformed moving image [16]. As proposed in [7], a zero-order B-spline Parzen window for the fixed image and a cubic B-spline Parzen window for the moving image is used. Together with a continuous cubic B-Spline representation of the floating image, this allows to calculate the gradient of the similarity metric S in closed form.

The metric S is computed as

$$S(\mu) = -\sum_{\tau}\sum_{\eta} p(\tau, \eta; \mu) log\left(\frac{p(\tau, \eta; \mu)}{p_{fix}(\tau; \mu) p_{mov}(\eta; \mu)}\right)$$

where p is the joint pdf with fixed image intensity values τ and warped moving image intensity values η. p_{fix} and p_{mov} represents marginal pdfs.

The derivative of S with respect to transformation parameter μ_i is following sum over all the fixed image voxels that are within support region of μ_i:

$$\frac{\partial S}{\partial \mu_i} = -\alpha \sum_{\mathbf{x}} \frac{\partial p(i_{fix}, i_{mov})}{\partial i_{mov}}\Big|_{i_{fix}=f_{fix}(\mathbf{x}), i_{mov}=f_{mov}(T(\mathbf{x};\mu))}$$

$$\cdot \left(\frac{\partial}{\partial \xi} f_{mov}(\xi)|_{\xi=T(\mathbf{x};\mu)}\right)^T \cdot \frac{\partial}{\partial \mu_i} T(\mathbf{x}; \mu)$$

where α is a normalization factor.

We use all the voxels of the fixed image to calculate the histogram and not only a subset like in [7]. We also use a multi-resolution approach to increase robustness and speed [16], meaning that we first register with reduced image and transformation resolutions, and then successively increase them until the desired resolution has been reached. We downsample the images by 2 in each dimension using gaussian filter and interpolation to obtain them in lower resolution.

3 GPU Architecture Overview

A GPU can be modeled as a set of SIMD multiprocessors (SMs) each consisting of a set of scalar processor cores (SPs). The SPs of a SM execute the same instruction simultaneously but on different data points. The GPU has a large device global memory with high bandwidth and high latency. In addition, each SM also contains a very fast, low-latency on-chip shared memory.

In the *CUDA* programming model [9], a *host* program runs on the CPU and launches a *kernel* program to be executed on the GPU device in parallel. The kernel executes as a grid of one or more thread blocks. Each thread block is dynamically scheduled to be executed on a single SM. The threads of a thread block cooperate with each other by synchronizing their execution and efficiently sharing resources on the SM such as shared memory and registers. Threads within a thread block gets executed on a SM in the scheduling units of 32 threads called a *warp*. Global memory is used most efficiently when multiple threads simultaneously access words from a contiguous aligned segment of memory, enabling GPU hardware to *coalesce* these memory accesses into a single memory transaction.

The Nvidia Tesla C1060 GPU used in the present work contains 4GB of off-chip device memory and 16KB of on-chip shared memory. The GPU supports a maximum of 512 threads per thread block.

4 Parallelization and Optimization

The main steps of the registration algorithm are outlined in Algorithm 1. The iterative gradient descent optimization part (within the first for-block) consumes almost all the computation of the registration algorithm. It has been shown that the two (for) loops that iterate over all the fixed image voxels take more than 99% of the optimization time [12]. We therefore focus our parallelization and optimization effort on this part following the CUDA programming model.

4.1 Parallel Execution on the GPU

We offload the two (for) loops to the GPU. Whenever the control on the host reaches any of these loops, the host calls the corresponding GPU routine. Once the GPU routine finishes, the control is returned to the host for further execution.

Data Partitioning and Distribution: To enable parallel computation of joint histogram (*loop 1*) and parallel computation of gradient (*loop 2*) on the GPU, we divide the images into contiguous cubic blocks before the start of iterative gradient descent optimization part and process the fixed image blocks independently. The fixed image blocks are distributed to the CUDA thread blocks with each thread block processing one or more image blocks. A thread block requires N/T iterations to process a fixed image block, where N is the number of voxels in the image block and T is the number of threads in a thread block. In each iteration, a thread in the thread block processes one voxel of the fixed image block performing operations in loop 1 or loop 2.

Algorithm 1. Image Registration Algorithm

Construct the image pyramid
For each level of the multi-resolution hierarchy
  Initialize the transformation coefficients
  For each iteration of the gradient descent optimizer
    Iterate over all fixed image voxels (loop 1)
      Map coordinates to moving image space
      Interpolate moving image intensity
      Update joint histogram
    Calculate joint and marginal probability density functions
    Calculate the mutual information
    Iterate over all fixed image voxels (loop 2)
      Map coordinates to moving image space
      Interpolate moving image intensity and its gradient
      Update the gradients for the coefficients in the neighborhood
    Calculate the new set of coefficients (gradient descent)

For example, for the fixed image block size of 8×8×8, we use 3 dimensional thread block of size $(8, 8, D_z)$. A thread block makes $8/D_z$ iterations for processing an image block, where D_z is the number of threads in the third dimension. In m^{th} $(0 \leq m < 8/D_z)$ iteration a thread with index (i, j, k) processes voxel $(i, j, k + m \times D_z)$ of the image block. As described in section 3, a maximum of 512 threads are supported per thread block and a single image block of size 8×8×8 also contains same number (512) of voxels. This allows a thread block with 512 threads to process one fixed image block in a single iteration. Moreover, as the fixed image blocks are stored contiguously in the global memory, the threads can read the consecutive fixed image values in a coalesced fashion.

The reference and moving images don't change as part of the iterative optimization steps. Therefore we transfer these images to the GPU global memory in the beginning and never modify them throughout the optimization process.

Joint Histogram Computation: The joint histogram computation in the first loop requires several threads to update (read and write) a common GPU global memory region allocated for the histogram. This will require synchronization among the threads. As described in section 2 we use parzen estimation of joint histogram with cubic B-spline window for moving image. This requires four bins to be updated with cubic B-spline weights for each pair of fixed image and warped moving image intensity values. The two threads with same intensity value pair will have collision for all of these four bins. Moreover, there will be collisions even if the threads have different intensity value pairs but they share some common bins to be updated. An atomic update based approach would be costly in this case and therefore we allocate a separate local buffer per CUDA thread in the GPU global memory to store its partial histogram results. A joint histogram with 32x32 (or 64x64) bins requires 4K (or 16K) memory for its storage therefore it is not possible to store these per thread partial histogram buffers in the on-chip

GPU shared memory. In the end, we reduce these buffers on GPU to compute the final joint histogram values.

Gradient Computation: Similar to the joint histogram computation, the gradient computation in second loop also requires several threads to update (read and write) a common GPU global memory region allocated for storing gradient values. Each thread processing a voxel updates gradient values for 192 (3×4×4×4) coefficients in the neighborhood affected by the voxel. Single precision gradient values for 192 coefficients require 768 bytes of memory and hence it is not possible to allocate per thread partial gradient buffer on the on-chip 16KB shared memory for more than 21 threads per thread block. Therefore, we allocate a separate local buffer per CUDA thread in the GPU global memory to store its partial gradient results. In the end, we reduce these buffers on GPU to compute the final gradient values.

For both Joint Histogram and Gradient Computation, the final reduction of partial buffers is performed using a binary tree based parallel reduction approach that uses shared memory [4].

Marginal pdfs and Mutual Information Computation: The computation of marginal pdfs for fixed and moving images together with mutual information computation is still done on the host. This computation on the host takes less than 0.02% of the total gradient descent optimization time and hence does not become a bottleneck. Performing this computation on the host requires transformation coefficients to be sent to the GPU before performing joint histogram computation. Once the computation is done, the computed joint histogram is transferred back to the host. Similarly, for the gradient computation, we send modified histogram values to the GPU before the computation and transfer back the computed gradient values to the host in the end. However, this transfer of data does not become the bottleneck as the amount of data transferred is small. The experiment results also show that these transfers require less than 0.1% of the total time for joint histogram and gradient computation.

4.2 Use of Look Up Table

When computing transformed fixed image voxels and moving image interpolation of transformed voxels, we need to perform weighted sum of B-spline coefficients with B-spline weights. As the cubic B-spline base functions only have limited support, therefore this weighted sum requires only four B-spline coefficients located at four neighboring control points. For 3D case, we need to consider only 4×4×4 points in the neighborhood of $\mathbf{x}$ to compute the interpolated value:

$$f(\mathbf{x}) = \sum_{i,j,k=0\ldots3} c_{i,j,k}\beta_{x,i}\beta_{y,j}\beta_{z,k}$$

where f is one component of the transformation function or moving image intensity, $c_{i,j,k}$ are B-spline coefficients and βs are cubic B-spline weights. For different components of the transformation function, weights remain the same but coefficients differ. Instead of computing these weights repeatedly at runtime,

we pre-compute these weights at sub-grid points with a spacing of 1/32 the voxel size and store the computed values to a lookup table. To enable the use of lookup table, we constrain the control point spacing to be an integral multiple of voxel spacing. Similarly, for image interpolation, we round down the point coordinates to the nearest sub-grid point.

We compute lookup table on the host and transfer it once to the GPU device memory along with the reference and moving images before the start of the optimization process. The lookup table doesn't get modified afterward and remains constant. For 32 sub-grid points in one voxel width, we only require 512 bytes of memory (four single precision B-spline weights per sub-grid point). On the GPU, we store the lookup table on the on-chip shared memory to avoid accessing high latency global memory each time.

4.3 Optimizations for Transformation Coefficients

As explained previously, each fixed image voxel requires 3×4×4×4 (4×4×4 per dimension) transformation coefficients in the neighborhood for transforming its coordinates to moving image space. However if the spacing between coefficient grid points is an integral multiple of fixed image block size, then all voxels within a fixed image block require same set of coefficients for the transformation. Each fixed image block is transformed by a single thread block and storing 3×4×4×4 coefficients only requires 768 bytes for single precision. Therefore, before processing an image block, all the 192 coefficients required by this block are loaded to the on-chip shared memory for faster access. There is no need for any synchronization in this case as threads only read the coefficients.

4.4 Memory Coalescing for the Gradient Computation

As described in section 4.1, for the joint histogram and gradient computation, we allocated per thread separate buffers in the global memory to store partial results. These buffers were reduced in the end to get the final values. For gradient computation, each voxel updates its local gradient buffer for its neighboring 4×4×4 coefficients per dimension. However, each thread of a warp updates the gradient buffer entry corresponding to the same coefficient in its local gradient buffer. The gradient buffer entries for the threads can be organized in either *Array of Structures* (AOS) or *Structure of Arrays* (SOA) form. In the AOS form, the gradient entries for a thread are stored contiguously in its separate distinct buffer. In the SOA form, the local buffers of different threads are interleaved so that gradient entries for different threads corresponding to same coefficient are stored contiguously in global memory.

In the AOS form, updates by threads result in non-coalesced memory accesses to global memory for read and writes. To avoid this, we use the SOA form for gradient computation. In this form, threads update consecutive memory locations in the global memory resulting in coalesced memory access. In the end, we reduce per coefficient values from all the threads to compute the final gradient values. We evaluate performance with these two forms in the result section 5.

5 Experimental Results

The parallel version of the code was run on the Nvidia Tesla C1060 GPU. The Tesla C1060 is organized as a set of 30 SMs each containing 8 SPs with a total of 240 scalar cores. The scalar cores run at the frequency of 1.3 GHz. The GPU has 4 GB of off-chip device memory with peak memory bandwidth of 102 GB/s. It has 16 KB of on-chip shared memory and 16K registers available per SM. The CUDA SDK 2.3 and NVCC 0.2.1221 compiler were used in the compilation of the code. The GPU host system has Intel Xeon 3.0 GHz processor with 3 GB of main memory. The GPU has a PCIe x16 link interface to the host providing a peak bandwidth of 4 GB/s in a single direction. The serial version of the code was running on one of the cores of an Intel Xeon processor running at 2.33 GHz with 2 GB of memory. The code was compiled with the GCC 4.1.1 compiler (with -O2). For both the systems, we measured the runtime of the most computation expensive multi-resolution iterative optimization process for single precision data. On the GPU system, it is assumed that the reference and moving images are already transferred to the GPU global memory.

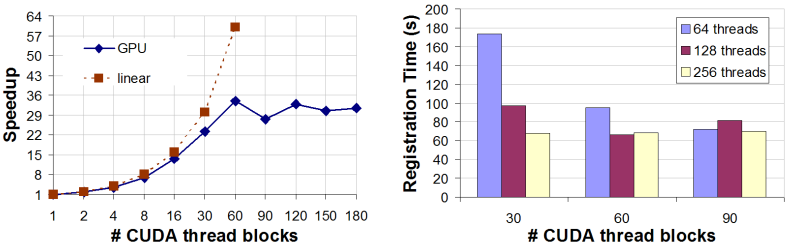
5.1 Performance Results and Discussion

Runtime. We performed registrations of 22 different sets (pair) of CT abdominal images of different sizes, and measured the average of the registration time per voxel. The images were partitioned into cubic blocks of size 8×8×8. A three level multi-resolution pyramid was used with B-spline grid spacing of 16x16x16 voxels at the finest level. The gradient descent optimizer was set to perform fixed number of 30 iterations at each pyramid level. For the purpose of comparison we measured the per voxel time for the finest pyramid level only. The sequential version required 69.02 (± 16) $\mu s/voxel$. In contrast, the GPU version required 2.46 (± 0.33) $\mu s/voxel$ resulting in factor of 28 speedup compared to the serial version. For example, the sequential version required 1736.65 seconds for an image of size 512x512x98 for 30 iterations at the finest level, whereas GPU version only required 67.20 seconds.

Note that the above time does not include the time for operations that are performed for each pyramid level before starting the iterations for the gradient descent optimizer e.g. allocating and transferring fixed and moving images on the GPU. To include these operations in the performance measurement as well, we compared the total registration application execution times for serial and parallel versions for all the 22 datasets. The GPU parallel based version showed a speedup between 18x to 26x with an average speedup of 22.3 (± 2.7) compared to serial version for total execution time. The good speedup suggests that the time consuming part of the application was successfully offloaded to the GPU.

Scalability. We measured the scalability of the GPU implementation with different number of threads per block and the number of thread blocks.

Figure 1(a) shows the scalability on GPU for an image of size 512×512×98 with different number of thread blocks up to 180 thread blocks. The performance with a single thread block is taken as the speedup of factor one. The number of



(a) Scaling with thread blocks (b) Scaling with threads per thread block

Fig. 1. Scaling on GPU with different number of threads and thread blocks

threads per thread block has been fixed to 128 for this experiment. As shown in the figure, we see good scalability with increasing number of thread blocks with factor 34 speedup with 60 thread blocks over single thread block performance. There is no performance improvement beyond 60 thread blocks. As described perviously, we use per thread local buffers for computing joint histogram and gradient values. These buffers need to be initialized to zero before performing the computation and need to be reduced in the end. This initialization and reduction time increases with increasing number of threads in the kernel grid and compensate any slight improvement in the actual computation time. For example, with 512×512×98 image dataset, the reduction time for joint histogram computation (and gradient computation) increases from about 1.26% to 3.42% (and about 1.1% to 2.66%) of the total histogram (and gradient) computation time when increasing number of thread blocks from 60 to 180.

Figure 1(b) shows the scaling for an image of size 512×512×98 with different number of threads per thread block for 30, 60 and 90 thread blocks. We could not use 512 threads per thread block due to register overflow. Also the implementation requires a minimum of 64 threads for 8×8×8 cubic image block size. We observe the best performance with 60 thread blocks and 128 threads. Although we show the scaling for only three sets of thread blocks, more number of threads per thread block seem to provide better performance in general as expected. Also, the performance difference with different number of threads decreases with increasing number of thread blocks as seen by the performance variation for different threads with 30 thread blocks compared to 90.

Optimizations. We compared the performance with AOS and SOA forms for gradient computation discussed in section 4.4. The table 1 shows the speedup with SOA form over the AOS form. The SOA form enabling coalesced memory accesses provides significant performance improvement over the AOS.

5.2 Validation of Registration Results

In this experiment, we validate the results of image registration for mono and multi-modal cases. We used the BrainWeb [1] simulated MRI volumes I_{T1} and

Table 1. Speedup with SOA over AOS for gradient computation

Algorithm Component	Speed up
Per iteration of gradient descent optimizer	7-8x
Total Gradient Computation (loop 2) Time (including data transfer and reduction)	10-12x
Gradient Computation Only (excluding reduction)	10-11x

Table 2. Similarity metric values for mono and multi-modal case

Modality	Mono		Multi	
Registered Images	T1 to T1	T1D to T1	T1 to T2	T1D to T2
Serial	-1.9443	-1.8914	-1.3975	-1.3743
GPU	-1.8246	-1.8426	-1.3452	-1.3590

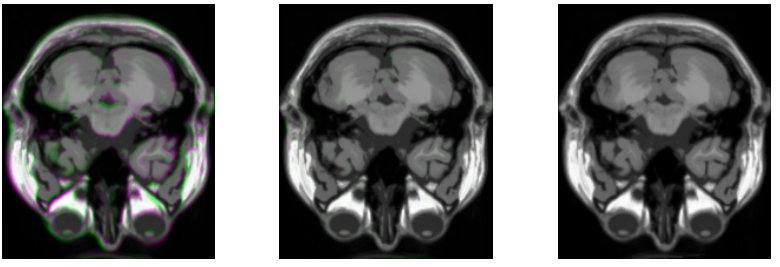
I_{T2} (181×217×181 voxels, isotropic voxel spacing of 1mm, sample slices shown in figures 3(a) and 3(b)): the T1 and T2 volumes are aligned. We deformed I_{T1} with an artificially generated transformation function T_D based on randomly placed Gaussian blobs to obtain I_{T1D}. We then registered I_{T1D} to I_{T1} (mono-modal case) and I_{T1D} to I_{T2} (multi-modal case) using both the serial and GPU implementations and measured the mutual information (MI) values before and after the registration. To verify the final MI values after the registration, we also compared these with the MI values of the perfectly aligned I_{T1} & I_{T1} (mono-modal) and I_{T1} & I_{T2} (multi-modal). We used three levels of the multi-resolution pyramid and 30 iterations were carried out per pyramid level. Table 2 shows the final values of similarity metric (negative of MI) after the registration along with the metric values for perfectly aligned images.

Mono-Modal Case: The similarity metric for perfectly aligned I_{T1} and I_{T1} is -1.9443 (computed using serial version). The final metric values after registering I_{T1D} to I_{T1} is -1.8914 and -1.8426 using the serial and GPU implementations respectively. The difference in metric values on the two platforms can be attributed to the difference in the floating point arithmetic and different ordering. The difference in the metric values for perfectly aligned images on the two platforms is similar.

Visual inspection of images before and after the registration confirmed that the registered image aligns well with the original image, and there is no difference between the images registered using the serial and GPU implementations. Figure 2 shows an example of visual comparison of the GPU registration accuracy.

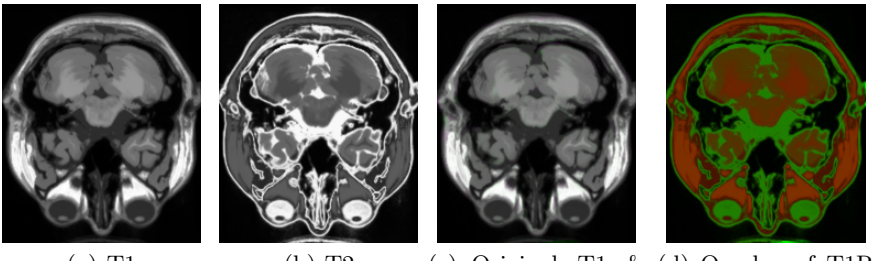
Multi-Modal Case: In this case, we registered I_{T1D} and I_{T2} to obtain registered image I_{T1R}. We then compared I_{T1R} to already known solution volume I_{T1} to verify the registration result. The similarity metric for perfectly aligned I_{T1} and I_{T2} is -1.3975. The final metric values after registering I_{T1D} to I_{T2} is -1.3743 and -1.3590 using the serial and GPU implementations respectively.

For visual comparison we cannot directly merge the registered image T1R with T2 as done in the mono-modal case. Therefore, we color-merged T1R and T1 for the purpose of validation. In case of correct registration, the color-merged image should not have any colored regions. Figure 3 shows the visual comparison of the GPU registration accuracy in case of multi-modal registration.



(a) Original and deformed (b) Original & registered image on GPU (c) Registered images on Xeon and GPU

Fig. 2. Visual comparison of the GPU registration accuracy in mono-modal case. Pairs of grayscale images are color-merged (one image in the green channel, the other in the red and the blue channel); areas with alignment errors appear colored. (a) shows the misalignment of the original (green) and the artificially deformed image. (b) shows that after registration on the GPU only minor registration errors are visible (original image in green). In (c), no difference between the image registered on the GPU and the Xeon (green) is visible.



(a) T1 (b) T2 (c) Original T1 & registered image (d) Overlay of T1R on T2

Fig. 3. Multi-modal case. (a) T1 image slice. (b) T2 image slice. (c) color-merged image of T1 (in green channel) and the registered image T1R on the GPU (in red and blue channels). Only minor registration errors are visible. (d) overlay of registered image T1R (red colored) on top of T2 (green colored) using 50% transparency.

6 Summary and Conclusions

We discussed in this paper a GPU-based implementation of mutual information based nonrigid registration. Our preliminary experimental results with the GPU implementation showed an average performance of 2.46 microseconds per voxel with 28x speedup over a serial version. For a pair of images of 512x512x24 pixels, the registration takes about 17.1 seconds to complete. Our GPU performance also compares well with the other high performance platforms, although it is difficult to make a perfectly fair comparison due to differences in implemented algorithms and experimental setup. A parallel implementation of mutual information based

nonrigid registration algorithm presented in [11] used 64 CPUs of a supercomputer reporting a speedup factor of up to 40 compared to a single CPU and resulting in mean execution time of 18.05 microseconds per voxel. The proposed GPU-based nonrigid registration provides a cost-effective alternative to existing implementations based on other more expensive parallel platforms. Future work will include more systematic and comparative evaluation of our GPU-based implementation vs others based on different multicore platforms.

References

1. Collins, D.L., Zijdenbos, A.P., Kollokian, V., Sled, J.G., Kabani, N.J., Holmes, C.J., Evans, A.C.: Design and construction of a realistic digital brain phantom. IEEE Trans. Med. Imaging 17(3), 463–468 (1998)
2. Courty, N., Hellier, P.: Accelerating 3D Non-Rigid Registration using Graphics Hardware. International Journal of Image and Graphics 8(1), 81–98 (2008)
3. Crum, W.R., Hartkens, T., Hill, D.L.G.: Non-rigid image registration: theory and practice. Br. J. Radiol. 77(2), 140–153 (2004)
4. Harris, M.: Optimizing parallel reduction in CUDA (2007), `http://www.nvidia.com/object/cuda_sample_advanced_topics.html`
5. Ino, F., Tanaka, Y., Hagihara, K., Kitaoka, H.: Performance study of nonrigid registration algorithm for investigating lung disease on clusters. In: Proc. PDCAT, pp. 820–825 (2005)
6. Kybic, J., Unser, M.: Fast parametric elastic image registration. IEEE Transactions on Image Processing 12(11), 1427–1442 (2003)
7. Mattes, D., Haynor, D., Vesselle, H., Lewellen, T., Eubank, W.: PET-CT image registration in the chest using free-form deformations. IEEE Trans. Med. Imag. 22(1), 120–128 (2003)
8. Muyan-Ozcelik, P., Owens, J.D., Xia, J., Samant, S.S.: Fast deformable registration on the GPU: A CUDA implementation of demons. In: ICCSA, pp. 223–233 (2008)
9. Nvidia CUDA Prog. Guide 2.3, `http://www.nvidia.com/object/cuda_get.html`
10. Pluim, J., Maintz, J., Viergever, M.: Mutual information based registration of medical images: a survey. IEEE Trans. Med. Imaging 22(8), 986–1004 (2003)
11. Rohlfing, T., Maurer Jr., C.R.: Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees. IEEE Trans. Inf. Technol. Biomed. 7(1), 16–25 (2003)
12. Rohrer, J., Gong, L., Székely, G.: Parallel Mutual Information Based 3D Non-Rigid Registration on a Multi-Core Platform. In: HPMICCAI workshop in conjunction with MICCAI (2008)
13. Rueckert, D., Sonoda, L.I., Hayes, C., Hill, D.L.G., Leach, M.O., Hawkes, D.J.: Nonrigid registration using free-form deformations: Application to breast MR images. IEEE Transactions on Medical Imaging 18(8), 712–721 (1999)
14. Sharp, G., Kandasamy, N., Singh, H., Folkert, M.: GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. Phys. Med. Biol. 52(19), 5771–5783 (2007)
15. Szeliski, R., Coughlan, J.: Spline-based image registration. Int. J. Comput. Vision 22(3), 199–218 (1997)
16. Thevenaz, P., Unser, M.: Spline pyramids for inter-modal image registration using mutual information. In: Proc. SPIE, vol. 3169, pp. 236–247 (1997)
17. Vetter, C., Guetter, C., Xu, C., Westermann, R.: Non-rigid multi-modal registration on the GPU. In: Proc. SPIE, vol. 6512 (2007)