Semi-Automatic Parallelisation for Iterative Image Registration with B-splines

Tamas Farago[1,2], Hristo Nikolov[2], Stefan Klein[3], Johan H.C. Reiber[1], and Marius Staring[1]*

[1] Division of Image Processing (LKEB), Department of Radiology, Leiden University Medical Center, Leiden, The Netherlands
[2] Leiden Institute of Advanced Computer Science, Leiden, The Netherlands
[3] Biomedical Imaging Group Rotterdam, Depts. of Radiology & Medical Informatics, Erasmus MC, Rotterdam, The Netherlands.

Abstract. Nonrigid image registration is an important, but resource demanding and time-consuming task in medical image analysis. This limits its application in time-critical clinical routines. In this paper we explore acceleration of a registration algorithm by means of parallel processing. The serial algorithm is analysed and automatically rewritten (re-coded) by a recently introduced automatic parallelisation tool, Daedalus. Daedalus identifies task parallelism (which is more difficult than data parallelism) and converts the serial algorithm to a Polyhedral Process Network (PPN). Each process node in the PPN corresponds to a task that is mapped to a separate thread (of the CPU, but possibly also GPU). The threads communicate via first-in-first-out (FIFO) buffers. Difficulties such as deadlocks, race conditions and synchronisation issues are automatically taken care of by Daedalus. Data-parallelism is not automatically recognised by Daedalus, but can be achieved by manually prefactoring the serial code to make data parallelism explicit. We evaluated the performance gain on a 4-core CPU and compared it to an OpenMP implementation, exploiting only data parallelism. A speedup factor of 3.4 was realised using Daedalus, versus 2.6 using OpenMP. The automated Daedalus approach seems thus a promising means of accelerating image registration based on task parallelisation.

Key words: image registration, parallel processing, Daedalus, CUDA

1 Introduction

Image registration is an important task in medical image processing. It refers to the process of spatially aligning data sets, possibly from different modalities, different time points, and/or different subjects [1, 2]. The application of nonrigid image registration tools in the clinic is limited by the consumption of time these algorithms typically exhibit. Applications such as image-guided surgery in the brain [3], where low quality intra-operative ultrasound scans need to be registered

* Corresponding author: `m.staring@lumc.nl`

to high quality pre-operative CT or MR scans, require the registration to be performed within a minute or preferably even less. Also in external radiotherapy there is a need for fast registration methods. Movements of organs may cause discrepancies between the expected radiation dose distribution and the actually received dose. Fast nonrigid registration would allow for on-line updating of the treatment plan [4, 5]. Currently, however, typical run times are ranging from 5 minutes to one hour or more, as reported by Klein *et al.* [6], owing to the large number of degrees of freedom that need to be estimated.

This paper aims at exploring techniques for accelerating image registration, by means of (semi-)automatic parallelisation. In general, moving from sequential computing to parallel computing is necessary because single-processor systems can not cope anymore with applications' complexity, throughput, and power consumption constraints that are inherent to so many applications. Although we are witnessing the emergence of parallel (multi-core and multi-processor) systems everywhere, the transition from sequential to parallel computing is far from trivial. Mapping sequential application specifications onto parallel systems is a difficult and time consuming task: the different tasks must be identified, after which they must be mapped onto different processing cores; proper synchronisation and data communication must ensure correct program execution. These complications pose a heavy burden on the developer and even upon success it is not guaranteed to get an increased performance.

As an alternative to manually re-coding the sequential registration algorithm, we investigated the automated parallelisation framework Daedalus [7, 8]. Daedalus identifies task parallelism (which is more difficult than data parallelism) and converts the serial algorithm to a Polyhedral Process Network (PPN). Each process node in the PPN corresponds to a task that is mapped to a separate thread of the CPU. The threads communicate via first-in-first-out (FIFO) buffers. Difficulties such as deadlocks, race conditions and synchronisation issues are taken care of by Daedalus. Section 2.2 explains the framework in detail. Data parallelism is not automatically recognised by Daedalus, but can be achieved by manually prefactoring the serial code to make data parallelism explicit, which is explained in Section 2.3. Experiments and results are presented in Section 3. For reference, the performance of a dedicated GPU implementation of the registration algorithm is also evaluated (Section 3.2).

2 Methods

2.1 Image registration framework

Image registration is defined as the problem of finding a spatial transformation $\boldsymbol{T}(\boldsymbol{x})$ relating two images of dimension d, one of which is fixed (I_F) and the other moving (I_M). In this paper, we focus on intensity-based image registration, formulated as an optimisation problem in which the cost function $\mathcal{C}$ is minimised with respect to the spatial transformation $\boldsymbol{T}$. The cost function defines the quality of the match. The framework is based on a parametric approach, meaning that the number of possible transformations is limited by introducing a parametrisation of the transformation. The optimisation problem reads:

$$\widehat{\boldsymbol{\mu}} = \arg\min_{\boldsymbol{\mu}} \mathcal{C}(\boldsymbol{T}_{\boldsymbol{\mu}}; I_F, I_M), \tag{1}$$

where the subscript $\boldsymbol{\mu}$ indicate the transformation parameters, a vector of size P. In the remainder of this paper the Mean Square Difference (MSD) metric is selected as a cost function:

$$\mathcal{C} = \mathrm{MSD}(\boldsymbol{T}_{\boldsymbol{\mu}}; I_F, I_M) = \frac{1}{N} \sum_{\boldsymbol{x} \in \Omega_F} \left(I_F(\boldsymbol{x}) - I_M(\boldsymbol{T}_{\boldsymbol{\mu}}(\boldsymbol{x}))\right)^2, \tag{2}$$

where Ω_F denotes the fixed image domain, and N the user-defined number of voxels sampled from Ω_F. An iterative optimisation routine is commonly used to solve (1), where we opt for a stochastic gradient descent approach [9]: $\boldsymbol{\mu}_{k+1} = \boldsymbol{\mu}_k - a_k \widetilde{\boldsymbol{g}}_k$, with k the iteration number, and with stop condition a maximum number of iterations K. The scalar a_k determines the step size, and is chosen as $a_k = a/(k+A)^\alpha$, with user-defined constants $a > 0$, $A \geq 1$, and $0 \leq \alpha \leq 1$; $\widetilde{\boldsymbol{g}}$ is an approximation of the cost function derivative $\partial\mathcal{C}/\partial\boldsymbol{\mu}$:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{\mu}} = \frac{-2}{N} \sum_{\boldsymbol{x} \in \Omega_F} \left(I_F(\boldsymbol{x}) - I_M(\boldsymbol{T}_{\boldsymbol{\mu}}(\boldsymbol{x}))\right) \left(\frac{\partial \boldsymbol{T}_{\boldsymbol{\mu}}}{\partial \boldsymbol{\mu}}(\boldsymbol{x})\right)^t \frac{\partial I_M}{\partial \boldsymbol{x}}(\boldsymbol{T}_{\boldsymbol{\mu}}(\boldsymbol{x})), \tag{3}$$

with $\partial\boldsymbol{T}_{\boldsymbol{\mu}}/\partial\boldsymbol{\mu}$ a matrix of size $d \times P$, $\partial I_M/\partial\boldsymbol{x}$ a vector of size d, and superscript t denoting the matrix transpose. The derivative is approximated by using a very small ($N \approx 2000$) subset of fixed image samples $\boldsymbol{x} \in \Omega_F$, randomly selected in every iteration k [9].

The transformation $\boldsymbol{T}_{\boldsymbol{\mu}}$ is in this paper the nonrigid 3rd order B-spline [10]. P is dependent on the chosen B-spline control point grid spacing and the image size, and can be anything from 10^3 to 10^6. The compact support of the B-spline basis function leads to a sparse Jacobian matrix $\partial\boldsymbol{T}_{\boldsymbol{\mu}}/\partial\boldsymbol{\mu}$, with only $d4^d$ nonzero entries. At the end of the registration the resulting image $I_M(\boldsymbol{T}(\boldsymbol{x}))$ needs to be computed, where 3rd order B-spline interpolation is used. This step is called resampling and can take several minutes on the CPU for large data sets. No multi-resolution strategies for the image data or the transformation are used in this paper. Algorithm 1 provides pseudo-code for the entire serial algorithm.

2.2 Automatic parallelisation

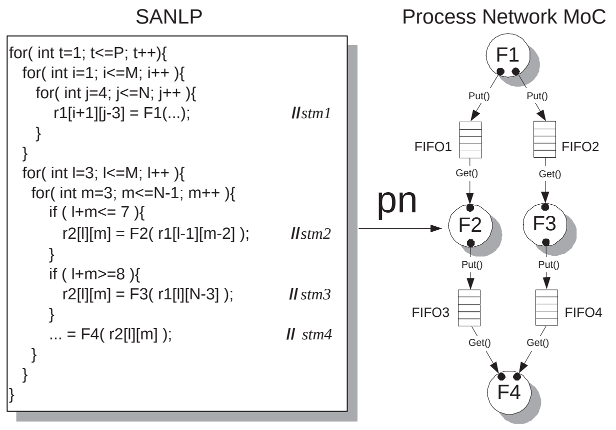
An important distinction when dealing with parallelism is that of data and task. Data parallelism refers to different processors performing the same task on different pieces of distributed data; task parallelism refers to each processor executing a different process on possibly different data. Both present possibilities for image registration. Data parallelism is commonly applied, with support readily available through special instruction sets (MMX, SSE), and standards extensions like OpenMP, MPI, and General Purpose GPU (GPGPU) programming. Task parallelism is in general more difficult to implement. Developers are

Algorithm 1 Pseudo-code for skeleton application

```
Require: $I_F, I_M, a, A, \alpha$, K
 1: for $k = 0$ to $K$ do
 2:     initialise $\mathcal{C}$ and $\partial\mathcal{C}/\partial\boldsymbol{\mu}$
 3:     $a_k \leftarrow a/(k+A)^\alpha$
 4:     get (random) samples from $I_F$
 5:     for $i = 0$ to $N$ do
 6:         $\boldsymbol{x}, f \leftarrow$ get coordinate and value of sample $i$
 7:         $\boldsymbol{y} \leftarrow \boldsymbol{T}_{\boldsymbol{\mu}}(\boldsymbol{x})$
 8:         $\boldsymbol{w} \leftarrow$ compute linear interpolation weights at $\boldsymbol{y}$
 9:         $m \leftarrow I_M(\boldsymbol{y})$ and $\boldsymbol{mx} \leftarrow \partial I_M/\partial\boldsymbol{x}(\boldsymbol{y})$ by linear interpolation
10:         $\boldsymbol{j} \leftarrow \partial\boldsymbol{T}_{\boldsymbol{\mu}}/\partial\boldsymbol{\mu}(\boldsymbol{x})$
11:         $\boldsymbol{jmx} \leftarrow$ get inner product of $\boldsymbol{j}$ and $\boldsymbol{mx}$
12:         $\mathcal{C}, \partial\mathcal{C}/\partial\boldsymbol{\mu} \leftarrow$ update value and derivative using (2) and (3)
13:     end for
14:     $\mathcal{C}, \partial\mathcal{C}/\partial\boldsymbol{\mu} \leftarrow$ finalise metric value and derivative
15:     $\boldsymbol{\mu}_{k+1} \leftarrow \boldsymbol{\mu}_k - a_k \cdot \partial\mathcal{C}/\partial\boldsymbol{\mu}$
16: end for
17: $I_M(\boldsymbol{T}_{\widehat{\boldsymbol{\mu}}}) \leftarrow$ compute the registration result with optimal parameters
```

not comfortable with the programming paradigm, few tools exist to assist, and the currently favoured implementation mechanism of using threads (alternatives exist, see [11]) is dangerous. A not-well written concurrent program has potential race conditions and deadlocks, and fixing them is extremely hard. Task parallelism lends itself very well to heterogeneous computing environments, where for example some threads run on the CPU and others on the GPU.

To facilitate the exploitation of task parallelism, the Daedalus framework [7, 8] (`http://daedalus.liacs.nl`) was proposed. Originally, it was aimed at embedded Multi-Processor Systems on Chip (MPSoC). Recently, Daedalus has been extended with a back-end towards heterogeneous desktop parallel computing (HDPC) [12], which generates code for a desktop computer. Starting from a sequential application specification in C, the open-source `pn` compiler [13] automatically converts it into a parallel Polyhedral Process Network (PPN) [14]. To enable the automation, the input source code is restricted to so-called *Static Affine Nested Loop Programs* (SANLPs), discussed in [13] (e.g. a conditional while-loop is not allowed, use a static for-loop instead). The PPN Model of Computation consists of autonomously running processes with distributed memory and control and communicate over FIFO channels using blocking FIFO read-/write primitives. The processes can execute on various computing devices such as the cores of the CPU, the FPGA, and/or GPU to take advantage of their respective strengths. For each process of a PPN, a thread on the host CPU is created. A core of a multi-core system is used for the actual computation when a process is assigned to it. For external devices, e.g., the GPU, the host CPU thread is only responsible for control flow, transfer of data to and from the device and controlling execution of the computation on the device. In summary, given a sequential program fulfilling the SANLP condition, the Daedalus framework



```
for( int t=1; t<=P; t++){
  for( int i=1; i<=M; i++ ){
    for( int j=4; j<=N; j++ ){
      r1[i+1][j-3] = F1(...);                    //stm1
    }
  }
  for( int l=3; l<=M; l++ ){
    for( int m=3; m<=N-1; m++ ){
      if ( l+m<= 7 ){
        r2[l][m] = F2( r1[l-1][m-2] );           //stm2
      }
      if ( l+m>=8 ){
        r2[l][m] = F3( r1[l][N-3] );             //stm3
      }
      ... = F4( r2[l][m] );                      // stm4
    }
  }
}
```

Fig. 1. Compiling a Static Affine Nested-Loop Program (SANLP) to a Polyhedral Process Network.

automatically identifies independent tasks and delivers a parallelised version, mapping each task to a processing node. Difficulties such as dead-locks, race-conditions, and synchronisation issues are automatically managed by Daedalus and can be safely ignored.

The derivation of a PPN from a SANLP is illustrated with an example in Figure 1. This example is taken from [15] that for the first time presented an analytical solution on how to extract PPNs from SANLPs. In Figure 1, a sequential program with 4 program statements is shown at the left-hand side. The functions read/write data only through affine array accesses. The derived and functionally equivalent PPN for this input code is shown at the right-hand side. Each program statement (stm$i$) is translated to one process (Fi), and the array accesses have been replaced such that the processes only communicate data over FIFO channels.

If a process node attempts to read data from an empty channel, the process is suspended until data is written into the channel. Similarly, if a process attempts to write data to a full channel, it is suspended until the FIFO channel has room for accepting the data. In HDPC, there are several mechanisms for realising the communication between the processes. For example, 'lock-free' channels are used for physically moving data in the computer memory. In contrast, 'acquire-release' mechanism exploits pointer arithmetic to avoid unnecessary data movement. Depending on the type of data, one approach is better than the other in terms of performance and communication overhead [12]. The buffer size of the channels can either be a minimal deadlock-free buffer, unbounded, or determined heuristically (smart). In the experiments, see Section 3.1, it is determined which setting is optimal for the particular problem of image registration.

A current limitation of Daedalus is that it does not automatically recognise opportunities for data parallelism. See for example the following code:

```
for ( int i = 0; i < N; i++ ){
  b(i) = F1( a(i) );                  //stm1
}
```

This code would result in a single process node `F1`, since there is only one statement, although there is an obvious opportunity for data parallelism. By slightly refactoring the code, the data parallelism can be exposed to DAEDALUS:

```
for ( int i = 0; i < N/2; i++ ){
  b(i) = F1( a(i) );                  //stm1
}
for ( int i = N/2; i < N; i++ ){
  b(i) = F1( a(i) );                  //stm2
}
```

This will result in two process nodes, each performing the same task. In order for DAEDALUS to support data parallelism, modifications to the `pn` compiler are needed.

2.3 Implementation

Algorithm 1 was implemented in C, fitting the SANLP requirements. This serial code serves as the baseline code, which is an input for DAEDALUS. Since DAEDALUS only determines task parallelism automatically, two manual modifications of the baseline code were made to exploit data parallelism. The loop over the samples $\boldsymbol{x}$ in the derivative (3) is independent from the sample number, and is therefore suitable for data parallelisation. Modification A divides this loop, see also steps 5 - 13 from Algorithm 1, in several (Z) independent chunks, see Section 2.2. This modification produces Z derivatives, one for each chunk, which have to be added together to form the final derivative of iteration k: $\partial\mathcal{C}/\partial\boldsymbol{\mu}_k = \sum_Z \partial\mathcal{C}/\partial\boldsymbol{\mu}_z$. Modification B additionally parallelises this addition.

2.4 GPU and OpenMP implementations

Much of the computation time for image registration is in the loop over the samples, steps 5 - 13 from Algorithm 1. This loop is a perfect candidate for data parallelisation: something a GPU is very capable of. Therefore, we investigate the suitability of the GPU for image registration by implementing Algorithm 1 in CUDA. For comparison, we also implemented data parallelism using the well-known OpenMP approach utilising the "`#pragma omp parallel`" statement.

For the GPU, computation of the transformation for each sample $\boldsymbol{T}(\boldsymbol{x})$ (used in step 7 and 17 of Algorithm 1) is implemented using the work of Ruijters *et al.* [16]. They decompose a 3$^{\text{rd}}$ order B-spline computation into a series of weighted linear interpolations, an operation which is hard-wired on the GPU using 3D textures. An issue with this approach is that the accuracy of the texture coordinates is limited. Therefore, a straightforward implementation circumventing this issue was also made available. Where Ruijters *et al.* use the decomposition

Table 1. Experimental details.

	fixed image size		moving image size		P
lung	124 164 187	$= 6 \times 10^5$	124 164 187	$= 6 \times 10^5$	2.4×10^4
small	52 75 165	$= 6 \times 10^5$	52 82 152	$= 6 \times 10^5$	3.4×10^5
middle	105 150 330	$= 5 \times 10^6$	105 165 305	$= 5 \times 10^6$	3.5×10^5
large	420 300 660	$= 8 \times 10^7$	420 330 610	$= 8 \times 10^7$	6.6×10^5

System 1	Intel Core2 Quad, 2.66GHz; Nvidia Quadro FX1700; WindowsXP 64bit
System 2	Intel Xeon W3520, 2.66GHz; Nvidia Geforce GTX285; Windows7 64bit
System 3	Intel Core i7, 2.6GHz; Nvidia Geforce GTX295; Windows Vista 64bit

for scalar interpolation, we extend it to the computation of transformations $\boldsymbol{T}$. This simply comes down to performing the operation for each dimension. Where at step 7 the transformation is computed only for $N \approx 2000$ samples, at step 17 it is computed for all voxels of the fixed image, a number typically in the range $10^6 - 10^8$.

3 Experiments and Results

Four 3D thoracic CT follow-up scans, varying in size (small, middle, large) have been used in the experiments, together with three different computer systems. Details are given in Table 1.

3.1 Daedalus registration results

Two important parameters of the Daedalus framework have been tested: the channel type (LF = lock-free, and AR = acquire-release channels), and the buffer size (MIN = minimal, and SMA = smart). Also the impact of modifications A and B (see Section 2.3) was evaluated. Next to the Daedalus implementation, the OpenMP implementation exploiting only data parallelism was tested. Non-rigid registration experiments were performed with the scan 'lung' on System 3, discarding resampling (steps 1-16 of Algorithm 1 only). See Table 2 for results.

In order to determine the maximum theoretical speedup, we ran several (independent) instances of the baseline code on System 3. The obtained performance results were 4.5x on the 4 core system using hyper-threading. This is an upper bound on the performance since there is no communication between the CPU cores. Daedalus with lock-free channels and minimal (PPN1a) and smart (PPN1b) buffers using the unadapted baseline code, results in a deterioration of performance, due to communication dominating the computation. Making data parallelism visible and using smart buffers (PPN2x) improves performance. PPN2a and PPN2c only use modification A (see Section 2.3); PPN2b and PPN2d additionally use modification B. PPN2a and PPN2b use lock-free channels; PPN2c and PPN2d use acquire-release channels, avoiding excessive data copying of $\boldsymbol{\mu}$ at the cost of some additional book-keeping.

Table 2. DAEDALUS results, all kernels executed on the CPU. Runtime in seconds. Registrations have been run using $N = 2048$ and $K = 1000$.

				Z or #threads					max.
method	chan.	buffer	modif.	1	2	4	8	16	speedup
Baseline				25.6	13.6	7.8	5.6	5.7	4.5
PPN1a	LF	MIN	-			120.1			0.2
PPN1b	LF	SMA	-			69.2			0.4
PPN2a	LF	SMA	A	19.0	20.2	20.5	20.8	22.6	1.4
PPN2b	LF	SMA	A&B	19.0	20.9	10.2	11.1	11.1	2.5
PPN2c	AR	SMA	A	15.7	17.0	16.6	18.0	19.8	1.6
PPN2d	AR	SMA	A&B	15.7	18.4	7.5	8.2	7.8	3.4
OpenMP				26.2	10.1	9.8	9.9	15.7	2.6

Making DAEDALUS aware of as much data parallelism as possible in combination with acquire-release channels (PPN2d) gives the best performance with a speed-up a factor of 3.4 on a 4 core machine. The DAEDALUS framework automatically exploits task parallelism, which is the most difficult to do manually, giving a gain of about 25%, compared to only exploiting data parallelism using OpenMP. The generated registration results are identical for all algorithms.

3.2 GPU registration and resampling results

The CUDA implementations of registration (steps 1 - 16) and of resampling (step 17) are tested on three datasets and compared with the CPU baseline implementation. Table 3 shows the timing results. For the registration, results are shown in the upper table. The results for registration were not exactly equal in terms of the final output $\widehat{\boldsymbol{\mu}}$ due to differences in the random number generator, but they were very similar. In Table 3 the gain in performance is reported. Speedup factors in the range 3.7 - 6.4 were measured.

As mentioned in Section 2.4, two implementations for the GPU were created: a straightforward implementation (GPU$_1$), and one based on the decomposition into linear textures (GPU$_2$). We generated a B-spline transform $\boldsymbol{T}'$ and applied it to the moving image. To validate the implementation, the Mean Square Difference (MSD) of the deformed moving image $I_M(\boldsymbol{T}'(\boldsymbol{x}))$ compared to the CPU-based result was measured. Resampling on the CPU was implemented in a multi-threaded fashion using data parallelisation. The bottom part of Table 3 shows the results. For resampling with the fast implementation GPU$_2$ factors of 10 - 65 were measured. Implementation GPU$_1$ is very accurate, the differences are all found at the boundary of the support region of the transformation, due to the use of a different boundary condition. Implementation GPU$_2$ has inaccuracies at sharp edges, for example at the interface of bone and soft tissue.

Table 3. GPU results for registration (top) and resampling (bottom). Timings are shown in seconds. top: $K = 1000$. Failure on System 1 was due to insufficient memory for the large data sets. Bottom: the last two columns show the mean square difference in result between CPU and GPU implementation.

		System 1			System 2		
image	N	CPU	GPU	ratio	CPU	GPU	ratio
small	2×10^3	21.4	10.9	2.0	16.1	4.3	3.7
	2×10^4	211.2	90.3	2.3	153.3	24.6	6.2
middle	2×10^3	23.1	11.0	2.1	17.2	3.9	4.5
	2×10^4	213.5	89.0	2.4	157.5	24.7	6.4
large	2×10^3	53.2	fail	-	32.8	7.1	4.6
	2×10^4	245.5	fail	-	175.7	28.8	6.1

	System 1				System 2				MSD (HU)	
image	CPU	GPU_1	GPU_2	ratio	CPU	GPU_1	GPU_2	ratio	MSD_1	MSD_2
small	2.2	0.3	0.1	15.5	0.8	0.1	0.1	8.8	1.3	2.7
middle	17.2	2.3	1.0	18.0	6.7	0.3	0.2	30.5	0.3	1.0
large	267.2	28.9	7.5	35.8	106.1	3.2	1.6	64.8	0.0	0.3

4 Discussion and conclusion

We have investigated the framework DAEDALUS for its use in image registration. This generic architecture (of potential interest to other areas of medical image processing) successfully extracted independent processes in the registration algorithm in an automated fashion. DAEDALUS relieves the developer from identifying independent tasks, setting up different threads and distributing tasks to them, and additionally from concerns about dead-locks, race-conditions, etc, that hinder correct program execution.

Manual work was still required, first to specify the serial algorithm in terms of a SANLP, and then to assist in identifying data parallelism possibilities. These steps present room for improvement in the DAEDALUS framework, but are already now of a much simpler nature than taking everything in own hands.

The DAEDALUS approach resulted in a speed-up a factor of 3.4 on a 4-core CPU, compared to 2.6 when only exploiting data parallelism using OpenMP. A complete rewrite of the algorithm in CUDA gave a speed-up of about 4.5 on a Geforce GTX 285, but presented much more labour and an additional requirement for the programmer to understand the workings of the GPU. For resampling the GPU proves to be an efficient computing device delivering a speed-up of 10 - 65.[4]

Future work includes the study of new bottlenecks, improving DAEDALUS, minimising random memory accesses on the GPU needed for computing $\partial \boldsymbol{T}/\partial \boldsymbol{\mu}$,

[4] The GPU resampler has been integrated in elastix, an open source software package for image registration [17], and will be available in the upcoming 4.4 release.

and the extension of the registration algorithm to more advanced techniques including multi-resolution and a mutual information cost function.

In conclusion, DAEDALUS is a very useful assistant for improving the performance of image registration algorithms, so needed for real-time application of imagery in the operating room.

References

1. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. Medical Image Analysis **2**(1) (1998) 1 – 36
2. Hill, D., Batchelor, P.G., Holden, M., Hawkes, D.J.: Medical image registration. Physics in Medicine and Biology **46**(3) (2001) R1 – R45
3. Pennec, X., Cachier, P., Ayache, N.: Tracking brain deformations in time sequences of 3D US images. Pattern Recognit. Lett. **24**(4-5) (2003) 801–813
4. Fei, B., Duerk, J.L., Sodee, D.B., Wilson, D.L.: Semiautomatic nonrigid registration for the prostate and pelvic MR volumes. Academic Radiology **12**(7) (2005) 815 – 824
5. Kerkhof, E., van der Put, R., Raaymakers, B., *et al.*: Intrafraction motion in patients with cervical cancer: The benefit of soft tissue registration using MRI. Radiotherapy and Oncology **93**(1) (2009) 115 – 121
6. Klein, A., Andersson, J., Ardekani, B.A., *et al.*: Evaluation of 14 nonlinear deformation algorithms applied to human brain MRI registration. NeuroImage **46**(3) (2009) 786 – 802
7. Nikolov, H., Stefanov, T., Deprettere, E.: Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(3) (2008) 542 – 555
8. Nikolov, H., Thompson, M., Stefanov, T., *et al.*: Daedalus: Toward composable multimedia MP-SoC design. In: 45th ACM/IEEE Int. Design Automation Conference (DAC'08), Anaheim, USA (2008) 754 – 579
9. Klein, S., Staring, M., Pluim, J.P.W.: Evaluation of optimization methods for nonrigid medical image registration using mutual information and B-splines. IEEE Trans. Image Process. **16**(12) (2007) 2879 – 2890
10. Rueckert, D., Sonoda, L.I., Hayes, C., *et al.*: Nonrigid registration using free-form deformations: Application to breast MR images. IEEE Trans. Med. Imaging **18**(8) (1999) 712 – 721
11. Lee, E.A.: The problem with threads. Computer **39** (2006) 33 – 42
12. Farago, T., Nikolov, H., Deprettere, E.: A framework for heterogeneous desktop parallel computing. Master's thesis, Leiden University, LIACS (2008) Internal Technical Report 08-17.
13. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: a tool for improved derivation of process networks. EURASIP J. Embedded Syst. **2007**(1) (2007) 19 – 19
14. Bhattacharrya, S., Leupers, R., Takala, J., Deprettere, E., eds.: Polyhedral process networks. In: Handbook on signal processing systems. Springer (2010)
15. Turjan, A.: Compiling Nested Loop Programs to Process Networks. PhD thesis, Leiden University (2007)
16. Ruijters, D., ter Haar Romeny, B.M., Suetens, P.: Efficient GPU-based texture interpolation using uniform B-splines. J. Graphics Tools **13**(4) (2008) 61 – 69
17. Klein, S., Staring, M., Murphy, K., Viergever, M.A., Pluim, J.P.W.: elastix: a toolbox for intensity-based medical image registration. IEEE Trans. Med. Imaging **29**(1) (2010) 196 – 205