AIRWC : Accelerated Image Registration With CUDA

Richard Ansorge 1st August 2008

BSS Group, Cavendish Laboratory, University of Cambridge UK.

We report some initial results using an NVIDA 9600 GX card to assist with 3D Medical Image Registration.

Affine registration was performed between two 3D anatomical volume data sets (size 240x256x176 voxels). The registration algorithm seeks to find an affine transformation that maps a “source” volume onto a “target” volume so as to minimise a cost function calculated between the two. In practice the transformed source voxels straddle target voxels positions, so that interpolation of target voxel values is required. Typically trilinear interpolation is used.

1) The CUDA Kernel:

The CUDA Kernel runs asynchronously on the GPU and executes many data-parallel threads simultaneously. Here each thread processes all the z values for a single x-y position in the source volume. The target dataset is stored in device Texture memory and the affine transformation is stored in device Constant memory. For each source voxel a fixed affine transformation is performed and then a 3D trilinear interpolation between target voxels is made using a Texture memory lookup. A cost function contribution (e.g. absolute value of difference) is then calculated. The kernel code is given in the appendix.

Other cost functions could be implemented is the kernel, although, at present, those requiring image histogram operations (e.g. mutual information based) might not perform as well. This is because to the lack of atomic (single thread) operations in the current generation of NVIDIA cards. Atomic operations will be available in the next generation of cards.

2) The Host Code:

This loads the source and target volumes to the device (required only once at the beginning of registration). The program then optimises the affine transformation to as to minimise the cost function. A variety of standard methods can be used to carry out the minimisation. We have used a Simplex algorithm, based on the Numerical Recipes Amoeba code. (The actual code needed significant modifications to be compatible with the current version of the NVIDA development system).

An updated affine transformation matrix needs to be uploaded to the GPU on each iteration (12 words), the resulting cost function also needs to be downloaded. At present we download a partial cost function contribution from each thread and sum them on the CPU. A further (modest) speedup could be achieved by doing this summation on the GPU.

3) Results:

Each iteration took 6.7 ms. The complete 6-parameter registration required 354 iterations and took 2.39 seconds to complete. The complete program required and additional ~300 ms for overheads such as loading files from disk.

In comparison the same registration using FSL running on a 2.4 GHz Xeon Linux workstation required 270 seconds, corresponding to a speedup of 112. Images of typical slices are shown in the Figure 1.

A second (more typical) registration was carried out between two different subjects. A speedup of about 85 was obtained. The results are illustrated in Figure 2. In fig 2.E and 2.F the target image (fig 2.D) is shown in red and the transformed source image (fig 2.C) is shown in green, overlapping areas appear yellow. A normalised correlation based cost function was also implemented for this test.

A simple 3 parameter registration was tried replacing the CUDA kernel with equivalent Host code. This showed that the CUDA kernel was actually 300 times faster than host code.

4) Comments:

The GPU used here is costs about £100 and requires a PC with a modest 400 Watt power supply; the card runs at 1.65 GHz and has 8 multiprocessors each having 8 computing cores. High end NVIDIA cards with 128 or more cores should offer another factor of 2-4 in speed.

Our GPU registration program was very basic and did not have optimisations, such as initial downsizing the images. Thus the speedups of ~100 achieved are quite conservative. Tests suggest the kernel is 300 times faster than equivalent host code.

We now plan to implement more demanding non-linear local registration, for example using cubic B-splines. Such registrations may involve 1000's of control points and can require days to run on single CPU's.

The tests were run on a Windows XP desktop machine and the GPU was managing the system display concurrently with processing.

5) Acknowledgement:

I am grateful to my colleagues at the Wolfson Brain Imaging Centre for encouragement and helpful discussions.

Figure 1 same subject whole head registration:

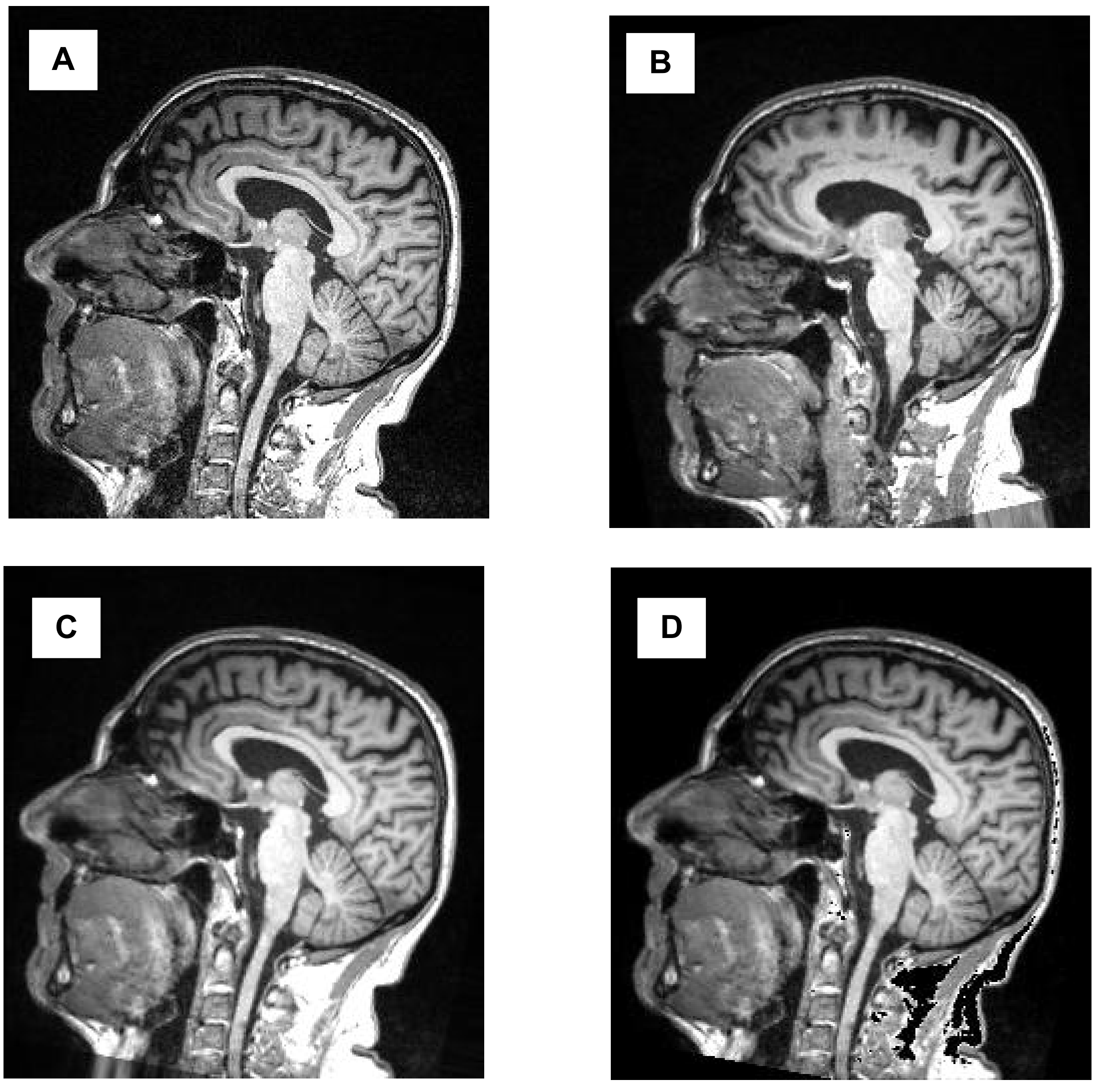

Figure 1: A: Slice from Target Image, B: Slice from Source Image in approximately similar position. The source image was manufactured by affine transformation and trilinear interpolation of the target image. C: Slice from Source Image, corresponding to fig A, after registration on GPU and second trilinear transformation back to original frame. D: Same as C but using FSL 4.1 on Linux workstation for registration.

Figure 2 Inter Subject Registration – after skull stripping.

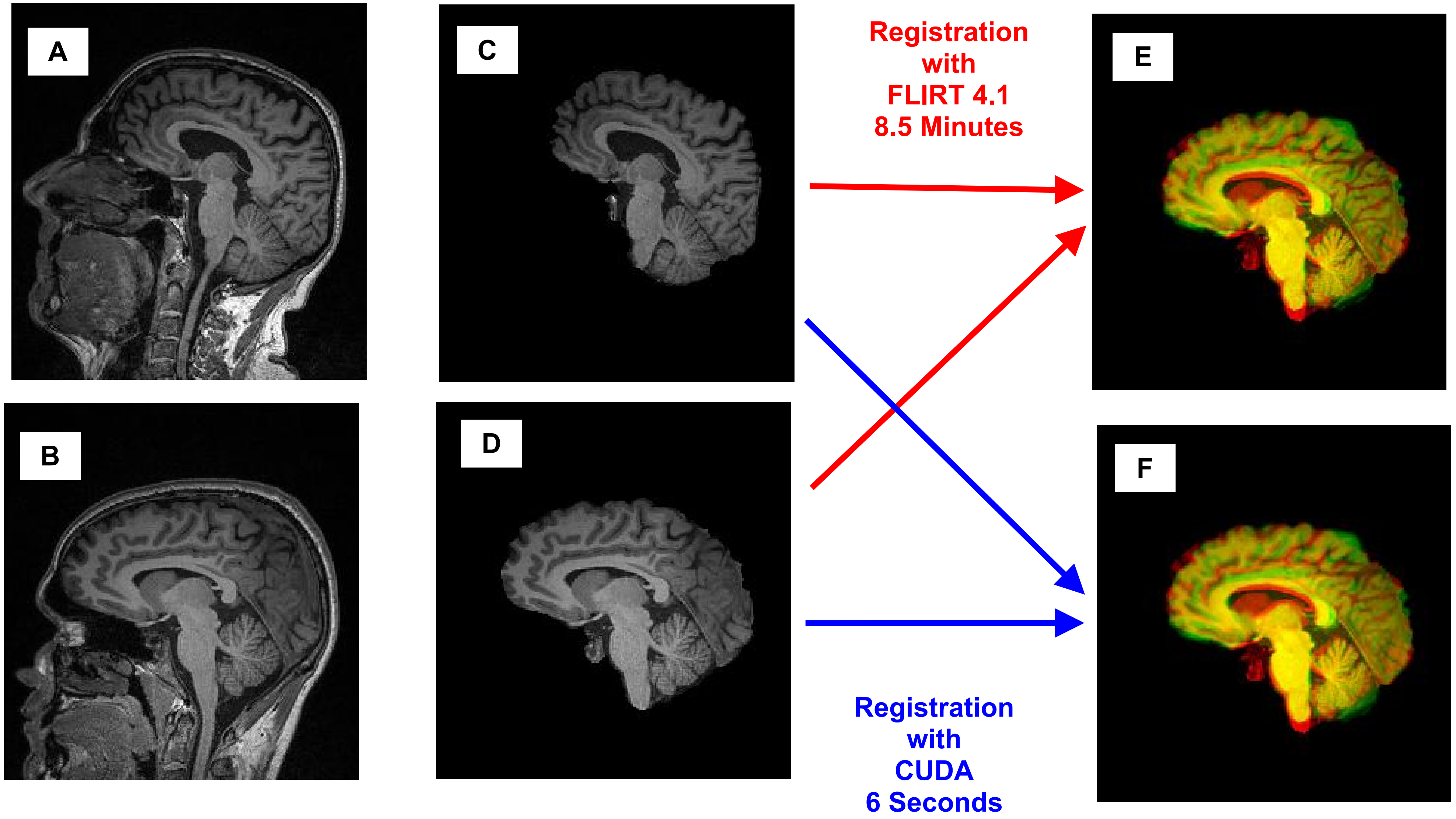
A
B
C
D
Registration with FLIRT 4.1 8.5 Minutes
E
F
Registration with CUDA 6 Seconds

Figure 2: A & B MRI scans at roughly corresponding slices for two different subjects, C &D same after "brain extraction". E 12 parameter affine registration of C to D with FSL 4.1, time taken 8.5 minutes, F same as E but using CUDA based AIRWC code, time taken 6 seconds.

Appendix Code for the Registration Kernel in CUDA 2.0 C++

```
// Affine registration kernel - July 2008
// Richard Ansorge (rea1@cam.ac.uk) University of Cambridge, Dept of Physics
// If copied please acknowledge the author

#include <cutil_math.h>

texture<float, 3, cudaReadModeElementType> tex1; // Target Image in texture
__constant__ float c_aff[16];                    // 4x4 Affine transform

// Function arguments are image dimensions and pointers to ouput buffer b
// and Source Image s. These buffers are in device memory

__global__ void d_costfun(int nx,int ny,int nz,float *b,float *s)
{
       int ix = blockIdx.x*blockDim.x + threadIdx.x; // Thread ID matches
       int iy = blockIdx.y*blockDim.y + threadIdx.y; // Source Image x-y

       float x = (float)ix;
       float y = (float)iy;
       float z = 0.0f;                     // start with slice zero
       float4 v =  make_float4(x,y,z,1.0f);
       float4 r0 = make_float4(c_aff[ 0],c_aff[ 1],c_aff[ 2],c_aff[ 3]);
       float4 r1 = make_float4(c_aff[ 4],c_aff[ 5],c_aff[ 6],c_aff[ 7]);
       float4 r2 = make_float4(c_aff[ 8],c_aff[ 9],c_aff[10],c_aff[11]);
       float4 r3 = make_float4(c_aff[12],c_aff[13],c_aff[14],c_aff[15]); // 0,0,0,1?

       float tx =  dot(r0,v);  // Matrix Multiply using dot products
       float ty =  dot(r1,v);
       float tz =  dot(r2,v);

       float source = 0.0f;
       float target = 0.0f;
       float cost = 0.0f;

       v.z=0.0f;
       uint is = iy*nx+ix;
       uint istep = nx*ny;
       for(int iz=0;iz<nz;iz++) {  // process all z's in same thread here
              source = s[is];
              target = tex3D(tex1, tx, ty, tz);
              is += istep;
              v.z += 1.0f;
              tx =  dot(r0,v);
              ty =  dot(r1,v);
              tz =  dot(r2,v);
              cost += fabs(source-target); // other costfuns here as required
       }
       b[iy*nx+ix]=cost;  // store thread sum for host
}
```

For nx=240, ny=256, nz=176 we use 15x16 thread-blocks each of size 16x16.