Freesurfer on the GPU

Richard Edgar

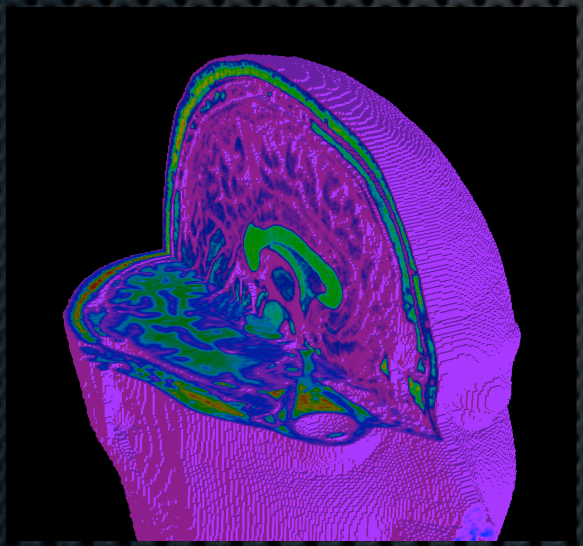
Overview

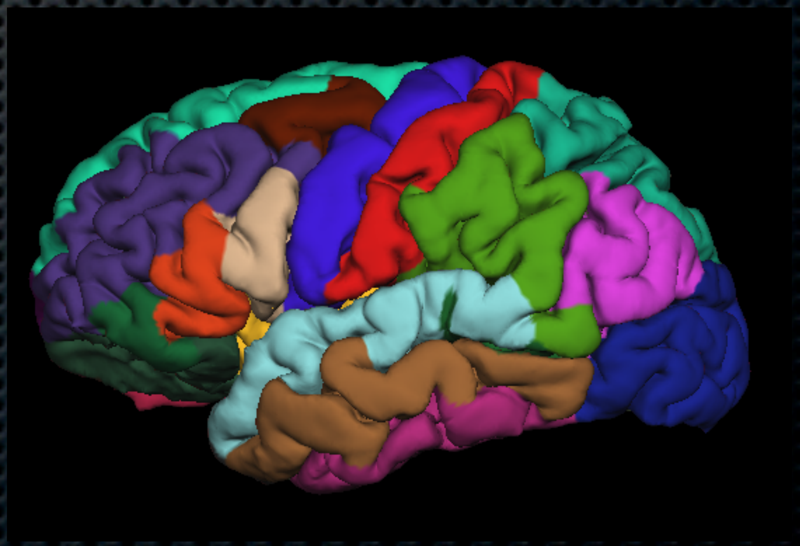
- What is Freesurfer?
- The Need for Speed
- Linear Registration
- Non-Linear Registration
- Future Needs & Directions
- Conclusion

The Freesurfer Suite

Freesurfer

- Set of tools for MRI brain image analysis
http://surfer.nmr.mgh.harvard.edu/
- Automatic registration & segmentation of images
- Around 1.3M lines of code





Usage

- Freesurfer used by thousands of researchers worldwide
  - Alzheimers
  - Aspergers
  - etc.
- Used in tandem with other techniques
  - EEG, MEG etc.
- Supported on multiple computing platforms

The Need for Speed

Key Driver

- Clinicians would like to use Freesurfer
  - Could help their diagnostics
- Need fast turnaround
  - Within an hour, or second visit required
- Main Freesurfer pipeline takes 10 hours
  - 3.2 GHz Intel W5580 (Gainestown/Nehalem)

Other Benefits

- ‘Quick’ registration while subject in MRI machine
  - Allows better targeting of fMRI/spectroscopy
- Faster population studies

Linear Registration

Linear Registration

- Task Outline
  - Take MRI image and precomputed atlas
  - Find affine transformation for best match
- Why accelerate first?
  - Key task
  - 20 minute runtime
  - Algorithm fairly simple

Basic Algorithm

- Pick affine transformation, A
- Evaluate total ‘energy’ for O(2000) atlas points
- Repeat, seeking lower energy

$$\begin{aligned} E(A) &= \sum_i f(y_i) \\ y_i &= Ax_i \end{aligned}$$

Multiscale Search

- Generate A from a base transform T
- Combine with small transforms $\{U_j\}$ $$A_j = TU_j$$
- Find the best A from this set
- This becomes the new T
- Generate new set of smaller U_j

Multiscale Search Example

- Start with identity transform
- Generate translations in range [-5,5]
  - Three in each direction
- Evaluate energies
- Select new T

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiscale Search Example

- Start with identity transform
- Generate translations in range [-5,5]
  - Three in each direction
- Evaluate energies
- Select new T

$$U_0 = \begin{pmatrix} 1 & 0 & 0 & -5 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$U_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\vdots$$

$$U_{26} = \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiscale Search Example

- Start with identity transform
- Generate translations in range [-5,5]
  - Three in each direction
- Evaluate energies
- Select new T

$$
\begin{aligned}
E_0 &= 500 \\
E_1 &= 493 \\
&\vdots \\
E_{26} &= 619
\end{aligned}
$$

Multiscale Search Example

- Start with identity transform
- Generate translations in range [-5,5]
  - Three in each direction
- Evaluate energies
- Select new T

$$T = U_2 = \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiscale Search

- Search performed in two stages
  - Translation only
  - Translation, rotation and dilation
- Each set U_j is a hypercube of possibilities
  - e.g. 5 possible translations in each direction etc.

First Acceleration Attempt

- Energy evaluation smallest parallel part
  - Evaluate each atlas point energy by one thread
  - Store results in global memory
  - Reduction sum to get total energy
- Transformation matrix sent from CPU

First Acceleration Attempt

- Atlas and MRI never change
  - Load at start of program
- Use texture for MRI
  - Free interpolation on co-ordinate transform
- Create GPU classes
- Use thrust library for reduction

First Acceleration Attempt

- Results promising
  - Runtime reduced to 4 mins (5x) with C2050
  - Transform identical
- However 2000 threads not much
  - Lots of performance still available

Increasing Parallelism

$$A_j = TU_j$$

- Energy evaluation for each A_j is independent
- Each U_j easily computed
  - Combination of translations, rotations and dilations
  - Parameters set by location in hypercube

$$\{U_j\} = \{D_\nu\} \otimes \{R^x_\mu\} \otimes \{R^y_\sigma\} \otimes \{R^z_\upsilon\} \otimes \{S_\eta\}$$

Block Indexed Transforms

$$A_j = TU_j$$

- Matrix T sent from CPU
- Compute transform set for U_j from block index
- First warp computes A_j and stores in shared memory
- Compute/reduce E_j in shared memory
- Store to global array

$$\{U_j\} = \{D_\nu\} \otimes \{R^x_\mu\} \otimes \{R^y_\sigma\} \otimes \{R^z_\upsilon\} \otimes \{S_\eta\}$$

Block Indexed Transforms

- Thrust selects minimum energy (and its index)
- Recover transform parameters from index
  - Use same routines as GPU
- Return to main program

Increasing Parallelism

- Now have hundreds of thread blocks
  - 5 translations in each direction gives 125 blocks
- Much better for the GPU
- Runtime 30s (40x) on C2050
  - Amdahl's Law now limiting factor

Analysing Results

- Speed up good, but results can differ
- Consider computation of A_j $$A_j = TU_j$$
  - First version computed on CPU and sent to GPU
  - Faster version computes A_j on GPU
- This gives slightly different results

Computation of A_j

- Actually use A_j^{-1}, not A_j
- First version inverts on CPU and sends that
- Faster version
  - Inverts T on CPU, sends to GPU
  - Trivially inverts components of U_j on GPU
  - Composes A_j^{-1} on GPU

$$\{U_j^{-1}\} = \{S_\eta\}^{-1} \otimes \{R_v^z\}^{-1} \otimes \{R_\sigma^y\}^{-1} \otimes \{R_\mu^x\}^{-1} \otimes \{D_\nu\}^{-1}$$

Computation of A_j

- Differences lead to different minimum
  - Occurs on subvoxel-sized transforms
  - End up with different final transform
- Assessing how to minimise differences

Non-Linear Registration

Motivation

- Linear registration insufficient
  - Diagnostics require detailed analysis of structures
  - Differences from atlas most interesting
- Require non-linear registration
  - Each voxel has own displacement

Non-Linear Registration

- Basic search algorithm similar
  - Pick a set of displacement vectors
  - Evaluate energy of configuration
- Dimensionality is millions
- Runtime over two hours
  - 3.2 GHz Intel W5580 (Gainestown/Nehalem)

Energy Evaluation

- Energy split into multiple terms $$E_{\text{tot}} = \sum_i \lambda_i E_i$$
- Each energy term follows same pattern
  - Evaluate expression for each voxel
  - Sum together
- General CUDA approach
  - Kernel to evaluate energies
  - Thrust for reduction sum

Transform Update

- Splits into multiple terms
  - Terms match energies
- Same pattern for evaluation
  - Each voxel produces new displacement vector
- CUDA acceleration follows same pattern

Converting to CUDA

- Basic prescription worked well
  - Most voxel evaluations independent
- Some floating point and precision issues
  - Able to keep within acceptable limits
- Datastructures were the main problem

Datastructure Conversion

- CPU code uses arrays of structures
  - Pointers to pointers
- 3D volumes use both xyz and zyx ordering
- None of this good for the GPU
  - Not so great for the CPU either

```
typedef struct {
  int width, height, depth;
  GCA_MORPH_NODE ***nodes;
  // .....
} GCA_MORPH;
```

```
typedef struct {
  double origx, origy, origz;
  // .....
  GCA* gc;
  // Total size 254 bytes
} GCA_MORPH_NODE;
```

Datastructure Conversion

- GPU required structure of arrays
  - Created templated ‘volume’ class to help
- Transfers between host and GPU very slow
  1. Allocate contiguous host arrays
  2. Pack data into these arrays (may have to reorder)
  3. Send across PCIe bus

Need for a Pipeline

- Datastructure conversion a significant bottleneck
  - CPU computation takes 200ms
  - GPU computation takes 20ms
  - Transfer back and forth takes 1s (round trip)
- Have to get entire computation on the GPU

Current Status

- All energy computations now pipelined on GPU
  - Runtime now around 90 minutes (C2050)
- Still working on the transform update
  - One major stage remaining
  - Datastructures even more interesting
  - Runtime <60 minutes looks possible

Future Needs & Directions

The Future is Hybrid

- Future machines will be hybrids
  - DARPA Exascale Computing Study
- Need programming paradigms to reflect this

Datastructures

- Rethinking of datastructures essential
  - Repacking stage kills performance
- Books teach arrays of structures
  - Nice way to think about things
- Performance requires structures of arrays
- How can we reconcile the two?

Datastructures

- Densely accessed structures are easy
- Create a class which
  - Holds separate arrays internally
  - Supplies operator() to construct individual instance
- Vector volume is an easy example

Datastructure Example

```
class VectorVolume {
public:
  float3 operator()( const unsigned int ix,
                     const unsigned int iy,
                     const unsigned int iz ) const {
    float3 res;
    res.x = this->x[ix + this->nx*(iy + this->ny*iz)];
    // etc.
    return( res );
  }

private:
  float *x, *y, *z;
  unsigned int nx, ny, nz;
};
```

Datastructures

- Sparsely accessed structures more difficult
- Only want to access required components
  - Loading full structure will hurt performance
- Compiler optimisations may help
  - Risky to rely on these

```
typedef struct {
  double origx, origy, origz;
  // .....
  GCA* gc;
  // Total size 254 bytes
} GCA_MORPH_NODE;
```

Datastructure Management

- Mentioned templated ‘volume’ class
- Actually two classes
  - ‘Management’ class for the CPU
  - ‘Mutator’ class for the GPU

Data Management Example

```
template<typename T>
class VolumeArgGPU {
public:
  const dim3 dims;

  __device__
  T operator()( const int ix,
                const int iy,
                const int iz ) const;
  // etc.

private:
  void* const pitchedPtr;
  const size_t dataPitch;
};
```

```
template<typename T>
class VolumeGPU {
public:
  operator VolumeArgGPU<T>( void ) const;

  void Allocate( const dim3 myDims );
  void Release( void );

  void SendBuffer( const T* const h_buffer );
  void RecvBuffer( T* const h_buffer ) const;
  // etc.

protected:
  dim3 dims;
  cudaPitchedPtr d_data;
};
```

Datastructure Management

- Useful wrapping of functionality on GPU
- Encourage on CPU too?
  - Separate management and computation
- Could define templated interface classes
  - CPU version could use same backend for both

Heterogeneous Computing

- Currently CPU and GPU structures separate
- Contain common metadata
  - e.g. Size of volume
- Can datastructures reflect this?
  - Currently have trouble keeping both updated

A Recipe for Heterogeneity

- Abstract base class defines
  - Metadata
  - Methods

```
class Image {
public:
  virtual void Allocate( const dim2 size ) = 0;
  virtual void Release( void ) = 0;
  // etc.
protected:
  dim2 imgSize;
};
```

A Recipe for Heterogeneity

- Subclass for specific hardware
  - Implement methods
  - Contain pointer to data

```
class ImageCPU : public Image {
public:
  // Implementations....
private:
  float* data;
};
```

```
class ImageGPU : public Image {
public:
  // Implementations....
private:
  float* d_data;
};
```

A Recipe for Heterogeneity

- Same with algorithms
  - Base class defines operation
  - Subclasses implement for hardware

```
class Convolve {
public:
  virtual void Convolve( const Image* src,
                         Image* dst,
                         const vector<float> kernel,
                         const char direction ) const = 0;
};
```

Heterogeneous Computing

- Main program only deals with base classes
- Supply conversion and assignment operators
- Freely mix code on different hardware

Conclusions

Conclusions

- Freesurfer can benefit greatly from GPUs
  - Linear registration as much as 40x
  - Non-linear registration now half an hour faster
- Need to consider structuring of future programs
  - Abstract implementation details
  - Provide high level interface to domain scientists

Acknowledgements

- Bruce Fischl
- Nick Schmansky
- Thomas Witzel
- Doug Greve
- Krish Subramaniam