A PARALLEL APPROACH TO DICCCOL

by

Michael Rob Lavoie

(Under the Direction of Tianming Liu)

Abstract

DICCCOL is a process that identifies common connectivity in the brain. It was developed to show that the cortex has a common structure thereby identifying functional correspondence. The tool compares the connectivity in a subject brain against a reference library of structural correspondence. A set of bundles is processed for comparison against this library. The result is a subject's fiber bundle that most closely matches the libraries reference bundle. The data set is relatively small but the processing is extensive. A single thread approach to the process is very time consuming. This task is better suited for a parallel processing approach. I show how the work can be accomplished more efficiently with GPU hardware and CUDA's parallel programming, resulting in a speedup factor of better than 6.

INDEX WORDS: DICCCOL, CUDA, GPGPU, eigenvector, eigenvalue, medical imaging

A PARALLEL APPROACH TO DICCCOL

by

Michael Rob Lavoie

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTERS OF SCIENCE

ATHENS, GA

2014

A PARALLEL APPROACH TO DICCCOL

by

Michael Rob Lavoie

Major Professor: Tianming Liu

Committee: Thiab Taha

Anthony Dickherber

Electronic Version Approved:

Julie Coffield

Interim Dean of Graduate School

The University of Georgia

August, 2014

ACKNOWLEDGEMENTS

I’d like to thank Professor Tianming Liu for his assistance with completing this work. I’d like to thank Dr. Kaiming Li for his assistance with, and his knowledge of the DICCCOL code base. I’d like to thank Jiang Xi for his assistance with the DTI data sets.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS IV

1. INTRODUCTION 1

2. RECENT AND RELATED GPGPU AND IMAGING WORK 7

3. MRI BACKGROUND INFORMATION 10

3.1 Nuclear Magnetic Resonance (NMR) 10
3.2 Magnetic Resonance Imaging (MRI) 17
3.3 Diffusion Weighted Image (DWI) 20
3.4 Diffusion Tensor Imaging (DTI) 25
3.5 Limitations of the process 30

4. THE CUDA PROCESS 32

4.1 Basic CPU-GPU architecture 32
4.2 GPU memory architecture 33
4.3 GPU programming model 37

5. CUDADOTRACEMAP PROCESS 44

5.1 Bundle To Covariance 45
5.2 Covariance Matrix To Principal Direction Vector 47
5.3 Principal Direction Vector To Feature Score 50
5.4 Bundle Concurrency 54
5.5 Reference Feature Data Correction 55
5.6 Polar Point Generation 56

6. TESTING 58

6.1 Eigenproblem Comparison 58
6.2 Serial vs Parallel Quantitative Comparison 58
6.3 Single ROI Visual Comparison 60
6.4 Multiple ROI Visual Comparison 60
6.5 Runtime Comparison 60

7. RESULTS 62

7.1 Eigen Calculation Accuracy 62
7.2 CPU vs GPU Bundle Results 62
7.3 CPU vs GPU Fiber Results 64

7.4 Fiber Segment Count Results 65
7.5 CPU vs GPU ROI-0 Visual Comparison Results 66
7.6 CPU vs GPU Single Subject Visual Comparison Results 73
7.7 Process Runtime Result 76
7.8 PredictROI2 Runtime Results 77
8. OBSERVATIONS 78
9. CONCLUSION 81
10. FUTURE WORK 83
11. MACHINE/SYSTEM DETAILS 87
ABBREVIATION GLOSSARY 88
REFERENCES 89

LIST OF EQUATIONS

Equation 1: Apparent Diffusion Coefficient Tensor for 3D space 26
Equation 2: Covariance calculation I,J=dimension ID, n=segment length..... 46
Equation 3: Eigenvector calculation 49
Equation 4: Feature Density calculation..... 52
Equation 5: Feature Density Normalization calculation..... 52
Equation 6: Polar Feature Count calculation 53
Equation 7: Feature Score calculation 54
Equation 8: Reference Library, feature correction calculation..... 55
Equation 9: Feature Density Normalization calculation..... 85

LIST OF FIGURES

Figure 1: Brodmann Map[29] 2
Figure 2: Cerebrum Cross-section[31] 3
Figure 3: Water Molecule [35] 11
Figure 4: Random Spinning Hydrogen Protons [35] 11
Figure 5: Proton Alignment in a Magnetic Field: High and Low Energy States [35] 12
Figure 6: Protons Precessing in a Magnetic Field [35] 12
Figure 7: Net Longitudinal Signal [35] (immeasurable) 13
Figure 8: Net Zero Magnetic Moment [35] 13
Figure 9: Net Transverse Signal: Phase Synchronized Precession [35] (measurable) 14
Figure 10: Long Chain Fatty Acid [35] 15
Figure 11: Pulse Sequence for a T2 Weighted Image 16
Figure 12: Pulse Sequence for a T1 Weighted Image 16
Figure 13: T1 Weighted Image Acquisition [35] 17
Figure 14: The red coils produce the Z-Axis Magnetic Gradient: Slice Selection [35] 18
Figure 15: The red coils produce the Y-Axis Magnetic Gradient: Phase Encoding [35] 19
Figure 16: The red coils produce the X-Axis Magnetic Gradient: Frequency Encoding [35] 20
Figure 17: Random walk of a single molecule 21
Figure 18: Pulse Sequence for a Diffusion Weighted Image [33] 23
Figure 19: Diffusion Weighting Process [33], B is the magnetic field strength 24
Figure 20: Converting signal intensity to ADC [34] 25
Figure 21: Apparent Diffusion Coefficient Tensor of a Voxel [38] 26
Figure 22: Converting signal intensity to eigenvector axes [34] 28
Figure 23: Isotropic Tensor (right) [39] Anisotropic Tensor (left) [40] 29
Figure 24: Chaining ellipsoids to build fiber tracts 30
Figure 25: Diffusion Tensor Image [41] 31
Figure 26: Single CPU processor and the many GPU coprocessors 33
Figure 27: Global memory coalescing 35
Figure 28: Striped Distribution of Shared Memory 36
Figure 29: Grid is a group of blocks Block is a group of threads 37
Figure 30: Block to Streaming Multiprocessor Assignment 38
Figure 31: Bundle to Covariance Grid Breakdown 45
Figure 32: Fiber Segmentation 47
Figure 33: Covariance to Principal Direction Grid Breakdown 48
Figure 34: Major eigenvector calculation 49
Figure 35: Feature Score 51
Figure 36: Principal Direction to Feature Score Grid Breakdown 52

Figure 37: Feature score comparison of each Polar 54
Figure 38: Polar Locations calculation 57
Figure 39: Subject 1, ROI 0 66
Figure 40: Subject 12, ROI 0 73
Figure 41: 62 Unique Ringed Polars 84
Figure 42: 62 Unique Equidistant Polars 85

LIST OF TABLES

Table 1: Processing breakdown of the subject 9 example 6
Table 2: Average Runtime Calculation example, Subject 9 61
Table 3: Absolute error with the CULA generated results 62
Table 4: Comparing the CPU vs GPU bundle results 63
Table 5: Subject 9 fiber bundle comparison 64
Table 6: Segment Count of Fibers in Bundle 574, ROI 0, Subject 9 65
Table 7: DICCCOL Process Runtime Comparison 76
Table 8: DICCCOL Runtime Comparison 77

LIST OF CODE BLOCKS

Code block 1: Pseudo code for the predictROI2 process 5
Code block 2: CPU code for adding 3000 number 39
Code block 3: CPU & GPU code for adding 3000 numbers 42
Code block 4: Pseudo code of main CUDA process 44
Code block 5: Pseudo code of the GpuBundleToCovariance process 46
Code block 6: Pseudo code of the Gpu3CovarianceToPrincipalDir process 48
Code block 7: Pseudo code of the GpuPrincipalDirToFeatureScore process 50
Code block 8: Pseudo code of the summation reduction technique 56
Code block 9: Comparing Segment Counts 60
Code block 10: DICCCOL Command Pipeline (with timers) 61
Code block 11: Sample CPU Run of DICCCOL, Subject 9 61

1. Introduction

DICCCOL[1]: Dense Individualized and Common Connectivity-based Cortical Landmarks

One of the challenges in the field of neuroscience is identifying the specific functional regions of the brain. The Brodmann map (Brodmann 1909, see Figure 1) is a collection of 52 areas of the human cortex gives a general structure and organization of the cerebral cortex. The cerebral cortex, more commonly referred to as gray matter (see Figure 2), is the surface or outermost (2-4 mm thick)[30] layer of the brain. It contains the neurons that provide processing and cognition. The Brodmann map, serves as a general organization of the anatomy. Due to the variability of the human cortex, the field is still struggling with the ability to precisely identify corresponding locations in different subjects. An example would be to precisely locate the region that controls the eye lids amongst different subjects. What is needed is the ability to identify a group-wise common cortical structure. With this information we would be able to predict, with some precision, the common structure amongst other subjects. This is the purpose of the DICCCOL tool. The tool predicts the cortical correspondence between a subject brain and a reference library of common cortical structure points. The points are called DICCCOLs.

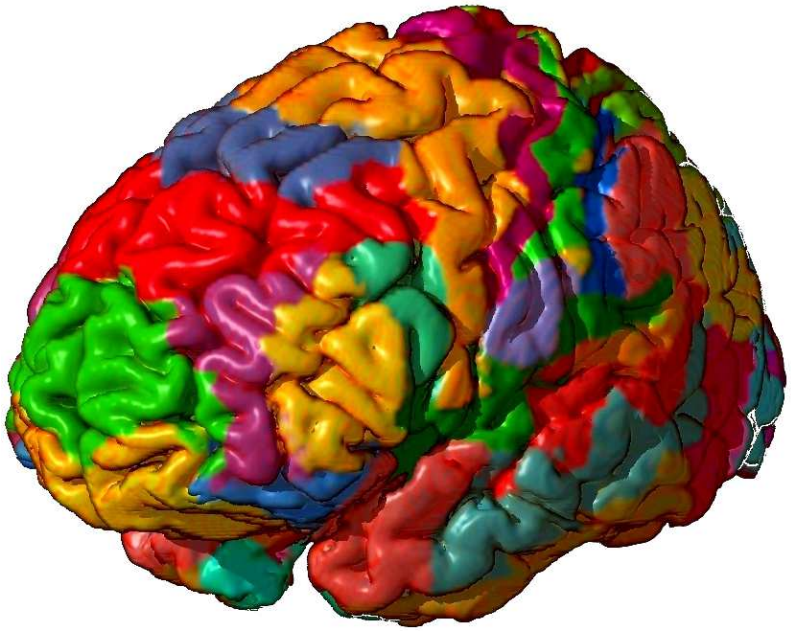


Figure 1: Brodmann Map[29]

Diffusion tensor imaging (DTI) is an MRI method that allows us to see the *vivo* fibrous structure of the cerebral cortex. It shows the axial fiber connectivity of the brain cortex, more commonly called white matter. This white matter is responsible for distributing and modulating the signals from the gray matter.[32] The white matter (see Figure 2), contains nerve fibers, or long axons, that interconnect different gray matter areas of the cerebrum.

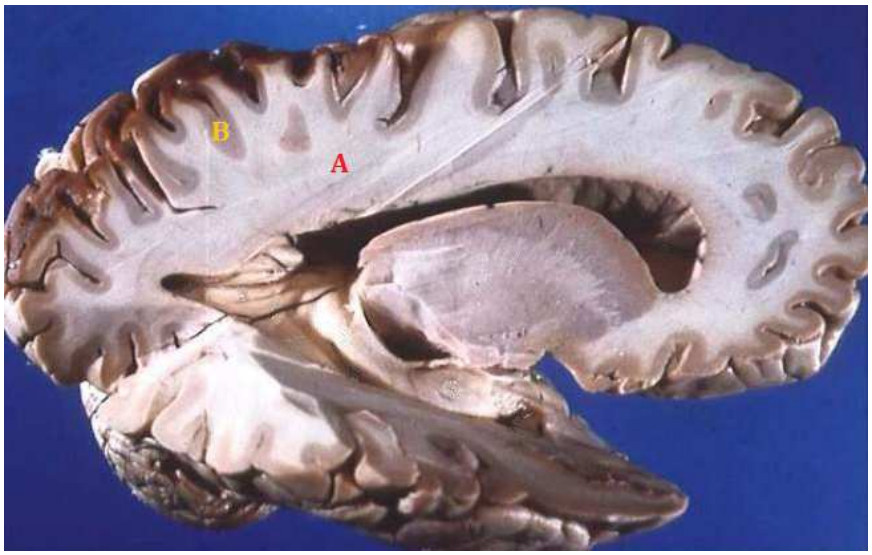


Figure 2: Cerebrum Cross-section[31] (A) White matter (B) Gray matter

By comparing a subject's connectivity against a set of DICCCOLs, we are able to locate and identify the corresponding landmarks between the two. DICCCOLs are a set of points that have shown dense correspondence amongst a variety of subjects.[1] The DICCCOL tool identifies this structural correspondence in the subject's cerebral cortex. With a subject's DTI scan, (the related fiber connectivity data), we can identify corresponding regions of common structural connectivity. The common connectivity is predicted by comparing the subject's brain connectivity against DICCCOL's reference library. The DICCCOL process can be characterized by three steps: Registration, Prediction, and the Viewer. The first step is the Registration. Here the FSL FLIRT[42] tool is used to register the subjects DTI scan against the library model. The registration results are used to adjust the subject's white-matter and fiber images to the shape of the library model. The predictROI2 tool then uses the registered images to locate a corresponding fiber bundle for the DICCCOLs in the library. Currently there are 358 landmarks in the DICCCOL library. The Viewer is the last step. Here the resulting subject bundle images and the reference library images are collected for display by the generateProfile4Viewer11 tool.

This interactive tool allows the user to rotate and view images of the subject cortex and fiber results, and the reference library.

By far, the most time consuming step of the full process is predictROI2. (See Code block 1, and Table 7) This is where I will focus my efforts to reducing the overall runtime of the DICCCOL tool. The process starts with a known DICCCOL landmark from the library. A similar landmark region is selected in the subject's brain. A set of fiber bundles in this region is selected for processing. Each of these different bundles is analyzed to identify the bundle with similar connectivity to that of the reference library landmark. Each fiber in the bundle is broken into overlapping segments. The Principal Direction of each segment is calculated then sorted according to direction. Once all the fibers in the bundle have been processed, the direction distribution percentages are compared against the landmark from the reference library. The process is repeated for all subject bundles in the region around the subject's landmark. The result yields the single bundle that best matches the reference landmark. This process is repeated for each DICCCOL in the reference library.

The DTI data set of fibers is relatively small and the process is very repetitive. It looks at the fixed set of DICCCOL landmarks. The neighborhood around each DICCCOL will generate a variable number of bundles. Each of these bundles will contain a variable number of fibers. Each fiber is divided into a variable number of fixed-length segments. The number of segments is dependent on the fiber length. These segments are sorted according to their spatial direction. The distribution of these directions is then compared against the reference DICCCOL to identify the bundle with the closest match.

```
predictROI2() {
    for(indxROI=0;  idxROI < 358;  idxROI++) {
```

```
        Get neighborhood of points around ROI
        for(idxNeighborhood=0;  idxNeighborhood<lastNeighborhood;  idxNeighborhood++) {
            Get fiber bundle for this neighborhood-point
            CudaDoTraceMap();
            Compare and save best-match-bundle result
        }
    }
}
```

Code block 1: Pseudo code for the predictROI2 process

There is no dependency between one fiber bundle and the next prospective bundle. Any single fiber may be part of more than one bundle. The data set for all bundles is known. The process essentially performs the same operation on many sets of data. This task is a prime candidate for parallel processing. Other than the single thread limit of the Central Processing Unit (CPU), there is no reason the bundles can't processed in parallel fashion. Rather than processing a single segment, a single fiber at a time, I should be able to process many segments from many fibers, all at the same time. A Graphics Process Unit (GPU) provides a SIMD approach, simultaneously executing many threads in parallel. I used an Nvidia GPU and CUDA programming to accomplish this parallel processing task. The CUDA code provides the common instructions that grids of parallel threads will execute. Rather than the original serial, single thread CPU approach, I will send the data to the GPU for parallel multithread processing. Once completed, the best-match result is sent back to the CPU and the process continues with the next DICCCOL.

Table 1 shows an example of the processing breakdown of a typical single subject. A single ROI would typically contain 20Mbytes of fiber data. The typical bundle size has 250Kbytes of data. The amount of repetitive processing of this data is quite extensive. The table is shown in a hierarchical format. The 358 DICCCOLs encompass 38,787 fiber bundles that need to be processed. These bundles contain 2million fibers. These fibers collectively contain 184million

fiber points. The majority of which are processed twice, resulting in 21million principal direction vectors.

Table 1: Processing breakdown of the subject 9 example

Number of DICCCOLs to process	358
Number of fiber bundles to process	38,787
Number of fibers to process	2,096,906
Number of fiber points to process	184,417,976
Number of trace points to process	20,955,341
RunTime of the serial (CPU) approach	11 minutes
RunTime of the parallel (GPU) approach	1.7 minutes

In this work I refer to the "GPU" as a heterogeneous CPU/GPU system, where the CPU is the host and the GPU is a coprocessor. The "CPU" refers to a CPU-only system.

To be consistent, the Subject9 data set is reference through most of this document. This subject was chosen at random from the available data sets.

2. Recent and related GPGPU and Imaging Work

Before the medical imaging and research field could make use of the GPU, they needed better hardware and software tools. The early GPU's used a graphics pipeline and fixed point numbers[7] rather than the double-precision floating point numbers and parallel thread approach used today. The early (2004-2007) programming tool Brook[8],[9],[12] used the graphics Application Programming Interface (API) (OpenGL, DirectX) to perform the General Purpose GPU (GPGPU) operations. The Nvidia hardware is supported by Compute Unified Device Architecture (CUDA). This is a proprietary C/C++ language (released 2007)[16], that supports a Single Instruction Multiple Data (SIMD) programming model. It interfaces with many other software packages. (Fortran, Ruby, Java, Perl, Python, MATLAB, Mathmatica, OpenCL, DirectCompute)[10] AMD's hardware is supported by the Close To Metal (CTM) proprietary programming interface. Their next generation (2010) Compute Abstract Layer (CAL), is supported by the Brook+[8],[12] extension, through the exposed Instruction Set Architecture (ISA). Intel doesn't offer a GPGPU solution. Their approach is to use their Many Integrated Core (MIC) technology to provide a High Performance Computing (HPC) solution. (Reference their Knights Corner and Knights Landing chips.)[43] The most popular GPGPU offering comes from Nvidia.[13] They have double-precision and Error Correcting Code (ECC) HW, and offer extensive software support. The lack of L2 cache on the early NVIDIA GPGPU's caused poor performance.[7] The Fermi and newer Kepler hardware offer both L2 cache and double precision capability.[13]

The amount of speedup a process might achieve depends on the application. This involves question such as: How large is the dataset? What amount of the dataset is reused? How extensive is the processing task?

A few CUDA reference works are as follows: An early (2001) example of GPGPU can be found in [7], where a matrix multiplication is demonstrated. An overview of CUDA and a few design considerations can be found in [12], [14] and [16]. Work load distribution in a heterogeneous GPU system is discussed in [15]. The GPU[11] is providing the necessary computing power that is showing speedup[12] in many basic computing tasks (*e.g.* quicksort, K-means, partial differential equations, linear algebra, sequence alignment).

In the medical imaging field, the computational performance of parallel processers is advancing development with visualization, segmentation, stereoscopic, and image analysis tasks.[11] Shane Ryoo[16] *et al.*, shows a CPU/GPU performance comparison of an MRI image reconstruction algorithm that shows significant speedup. Castaño-Díez[17] *et al.*, evaluated many common image processing algorithms (spatial transformations, real-space and Fourier operations, pattern recognition procedures, reconstruction algorithms, classification procedures). Their porting of the C code to CUDA saw a typical speedup of 10->20. The very large dataset generated by an fMRI modality requires a high degree of computational power. By applying the GPU to the fMRI process we can expect higher quality visualization, higher temporal & spatial resolution, and advanced real-time analysis. To wit, the Canonical Correction Analysis (CCA) and General Linear Model (GLM) statistical approaches have been implemented with MATLAB on a GPU.[18] The work showed a speedup of >10 over other MATLAB implementations. The work was also implemented in C, OpenMP, and CUDA, resulting in a speedup of >170 (GLM) and >800 (CCA) for a 1000 random permutation test.[21] A high quality white-matter fiber bundle visualization technique is demonstrated by Ottan *et al.* on a GPU using OpenGL.[19] Huang *et al.* offers a GPU replacement for Statistical Parametric Mapping (SPM) of the MRI automatic registration process. Their approach shows a speedup of 14 for single-modality intrasubject data

sets.[20] Brain perfusion quantification shows the cerebral blood flow (CBF), cerebral blood volume (CBV), and mean transit time (MTT) which can be used in the diagnosis of acute stroke. Quick diagnosis, (*i.e.* reduced analysis time) is essential for reducing the damage caused by a stroke. Zhu *et al.* offers a GPU approach that achieves a speedup (versus the CPU approach) of >5 for CT and >3 for MRI data sets.[22] (OpenMP and MPI routines are also tested.) A GPU approach has been used to dramatically improve performance in the Markov Chain Monte Carlo (MCMC) algorithm in the Bayesian Estimation of Diffusion Parameters Obtained using Sampling Techniques (BEDPOSTX)[25] toolbox of FMRIB Software Library (FSL).[24] The toolbox uses a diffusion-weighted magnetic resonance imaging (DW-MRI) data set to map the white matter connectivity of the brain. As imaging of neural tracts (tractography) algorithms evolve, they deliver finer details about the neural fiber pathways. The associated increase in complexity comes with increased computational cost. The CUDA approach developed by Hernández *et al.* produced a speedup of >100 on a single GPU system (versus a single CPU), and >120 for a multi-GPU system (versus multi-CPU system).[23] Lee *et al.* also offers a multi-GPU CUDA approach to the probabilistic BEDPOSTX algorithm. They develop and compare for two diffusion tensor based tractography algorithms. Their BEDPOSTX version shows a speedup factor of >60. The other, a deterministic Bayesian fiber tracking algorithm, shows a speedup factor of >100.[26] Wang *et al.* developed a hybrid multi-CPU-GPU system to accelerate the graph theoretical algorithms that are used in high-resolution functional brain network analysis (voxel based connectome).[27] A task that is highly computationally demanding, versus the more common (down-sampled fMRI data) low-resolution brain network approach. The largest speedup factor (>200) came from the All Pairs Shortest Path algorithm.

3. MRI Background Information

What follows is a general overview of the principals behind the science of MRI. There are many subtleties and approximations that are not mentioned here. A more in-depth discussion can be found in [33] and [34]. The words that follow attempt to define and describe the physics behind the NMR process, but words do not portray a level of understanding necessary to fully grasp the concepts presented here. The subject requires motion and animations to better grasp what is being explained. For an animated explanation of NMR principals see [35].

3.1 Nuclear Magnetic Resonance (NMR)

Nuclear Magnetic Resonance (NMR) combines physics principals of magnetism and Radio Frequency (RF) energy as they affect atomic nuclei. The origins of NMR date back to the 1930's and 1940's. RF energy refers to the 30KHz -300GHz frequency range of the electromagnetic spectrum. (Just above sound and just below infrared.) This frequency range includes: cell phones, wireless services, AM/FM radio, UHF/VHF TV, etc. An RF signal can be described by many properties. We are interested in: Frequency, Amplitude, and phase shift of an RF signal. Frequency is measured in cycles per second, or Hertz (Hz). The amplitude of an RF signal describes the strength of a signal; it is typically described in terms of Voltage (v). Phase shift describes a time shift between two signals; it is typically described in terms of angular degrees. For example, the typical home is powered with two electrical signals. Each signal has 110v of amplitude, running at 60Hz. The difference between the two signals is an 180^{o} phase shift.

Magnetism is measured in units of Tesla (T), named after Nikola Tesla, a pioneer in the field electromagnetics. A 1T magnetic field is generally considered a strong magnetic field. The earth's magnetic field is measured in micro Tesla (μT). A typical MRI machine generates a 3T magnetic field.

For this discussion, the chemical compound of interest is H_2O, the common water molecule, with a radius of ≈10μmeters[33]. (See Figure 3) By weight, the typical human is 60% water. The element we are interested in is the hydrogen (H) atom. Hydrogen is the lightest and simplest of elements. Its atomic weight is 1.00794u. (u= unified atomic mass unit= 1.660538×10^{-27}kgram)[44] The reason it is so light, it has only one electron (0.000910×10^{-27}kgram), one proton (1.672621×10^{-27}kgram), and no neutrons (1.674927×10^{-27}kgram). (If the atom had a neutron, the mass of the nucleus would double, making NMR much less likely.) We are interested in how the protons of a water molecule respond to a strong magnetic field and a burst/pulse of RF energy.

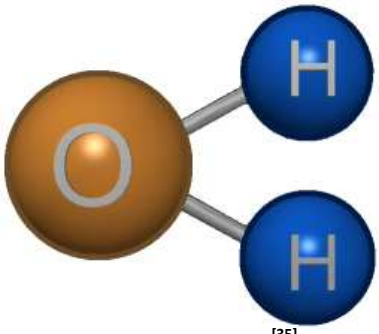


Figure 3: Water Molecule [35]

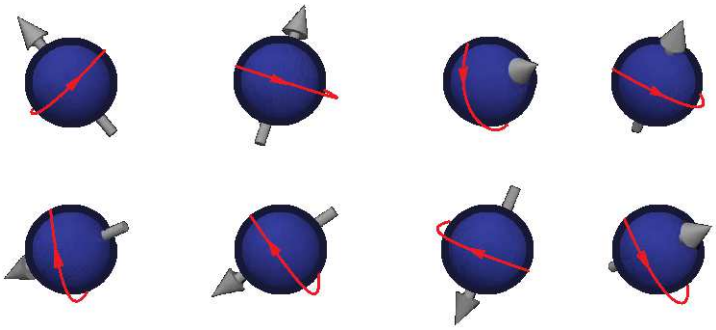


Figure 4: Random Spinning Hydrogen Protons [35]

As the negatively charged electron orbits the nucleus of the atom, the positively charged proton spins inside the nucleus. This subatomic spin generates a small magnetic field. Using the right hand, if we orient the fingers in the direction of rotation, the thumb will point in the direction of the magnetic field (or magnetic moment), generated by the spinning proton. In its natural state the protons randomly spin along no particular axis or direction, the net magnetic moment is zero.

In its natural state the human body has no magnetic moment, (it does not generate any significant net magnetic field). (See Figure 4)

When a strong enough magnetic field is applied to the body, it causes these random magnetic moments to lineup in the direction of the applied magnetic field. (See Figure 5) Most of the magnetic moments line up with the applied magnetic field, but some of the higher-energy protons, line up opposite to the applied magnetic field.

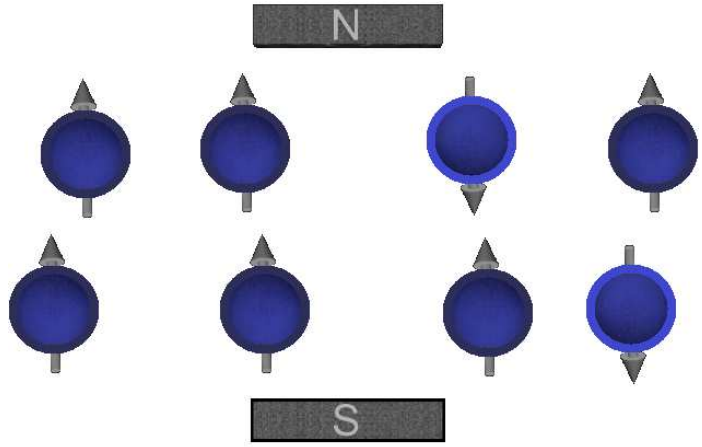


Figure 5: Proton Alignment in a Magnetic Field: High and Low Energy States [35]

These opposing-moment protons are said to be in a high-energy state, while the remaining protons are in a low-energy state.

While the applied magnetic field aligns the magnetic moments of the protons, it doesn't stop the protons from spinning. They continue to spin along the axis of the applied magnetic field. This spin action causes the protons to wobble in place, or precess. (See Figure 6)

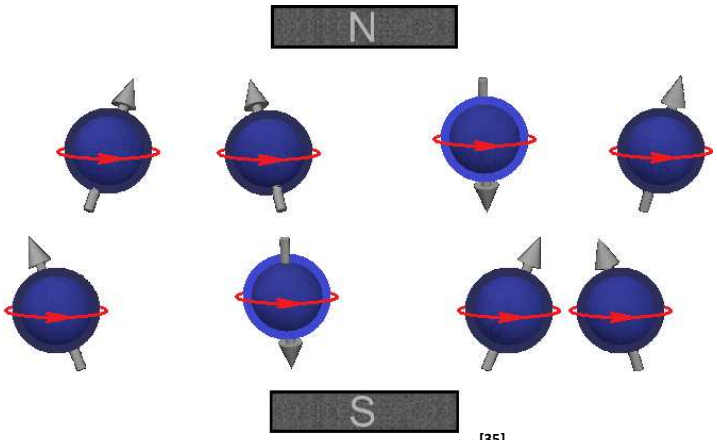


Figure 6: Protons Precessing in a Magnetic Field [35]

The rate of precession is related to the strength of the magnetic field. $f = \gamma B_0$, where $f =$ Larmor precession (resonant) frequency, $\gamma =$ gyromagnetic ratio (a constant), and $B_0 =$ strength of the applied magnetic field. In a 3T magnetic field, the Larmor frequency of a hydrogen proton is 127.74MHz.

We use this Larmor frequency to alter the energy state of the precessing protons. When a homogeneous magnetic field is applied to tissue, most of the hydrogen protons precess in the direction of the applied magnetic field. These low-energy state protons outnumber the high-energy state protons resulting in a net magnetization in the same direction as the applied magnetic field. Because this longitudinal magnetization runs in the same direction, it can't be separated from the applied field, and thus can't be measured. (See Figure 7)

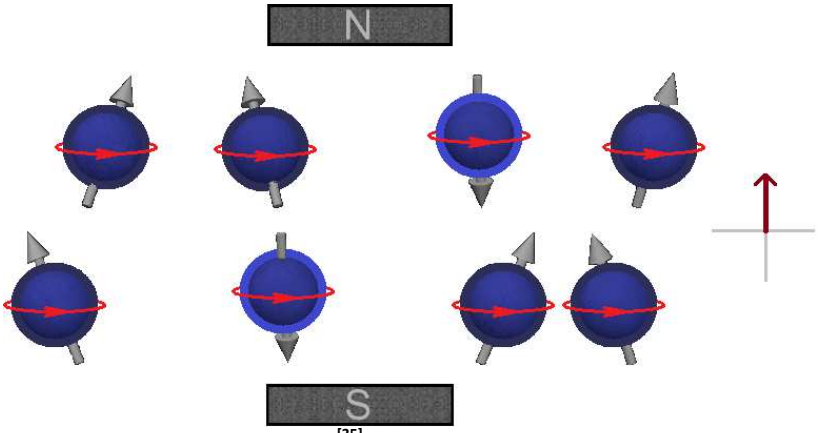


Figure 7: Net Longitudinal Signal [35] (immeasurable)

A significant burst of RF energy at the Larmor frequency will alter the energy state and phase of the precessing protons. This RF signal is applied perpendicular to the applied homogeneous

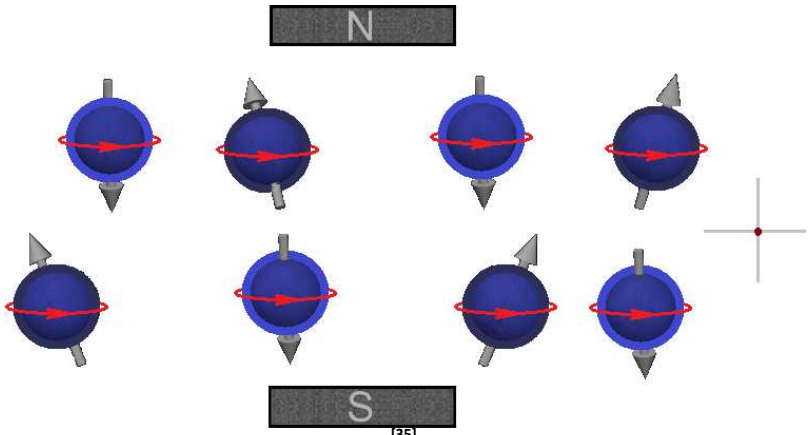


Figure 8: Net Zero Magnetic Moment [35]

magnetic field. The strength of this signal causes some of the low-energy state protons to flip to the high-energy state. When enough protons have flipped, the longitudinal magnetism is reduced to zero. (See Figure 8)

Now, half of the protons are now in a high-energy state, while the other 50% are in a low-energy state. The RF energy also causes the protons to wobble in phase. This phase synchronized precession produces a transverse magnetic signal that can be measured. (See Figure 9)

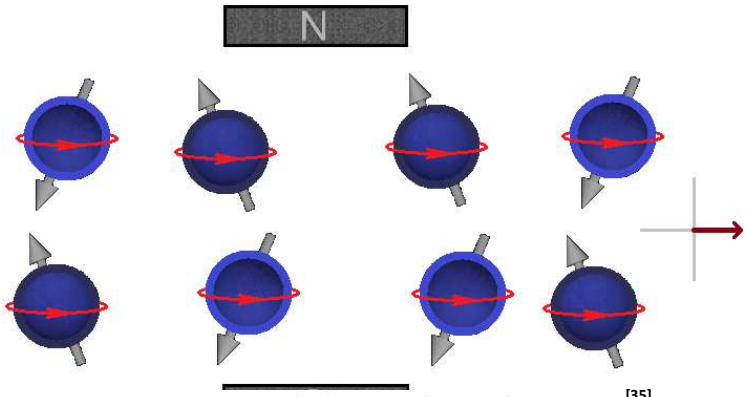


Figure 9: Net Transverse Signal: Phase Synchronized Precession [35] (measurable)

This transverse signal continues as long as the RF energy is applied. When it is removed, the phase alignment is lost due to the repellent force of all the positively charged protons. This process is called "T2 relaxation". This causes the transverse signal to reduce to zero. (See Figure 8) The protons that flipped energy state also give up their absorbed energy in the form of heat, and return to the low-energy state, thus recreating the net longitudinal magnetism. (See Figure 7) This process is referred to as "T1 relaxation".

In this discussion, we are primarily interested in the hydrogen atoms of the water molecule. There is nothing that prevents other hydrogen atoms in the tissue from responding to the applied magnetism and RF energy. In fact their response is the same, although time shifted. Water is a free roaming molecule when compared to a fixed-position fat protein. The density of hydrogen atoms in a fat molecule is much higher than that of the water molecule. (See Figure 10) The response of both of these molecules is the same when the magnetic field and RF pulse are applied. The difference is how quickly they respond when the RF pulse is removed. The relaxation occurs much faster in the hydrogen-dense fat molecule than the low-density water molecule. That is, the water molecule holds onto its high-energy state longer than the fat molecule. The difference is significant enough that it allows us to isolate the two different signals. As mentioned earlier, the NMR process only allows us to measure or observe the transverse magnetization signal. This signal is strongest just prior to the T2 relaxation, after which it degrades to zero, and then the immeasurable longitudinal signal is recreated. The transverse signal from the water molecules is readable because of their slow relaxation time and fat's much quicker relaxation time. While the water is beginning its relaxation process, the fat has finished its relaxation process. As a result, the water is generating a strong transverse signal, while the fat is generating little to no signal.

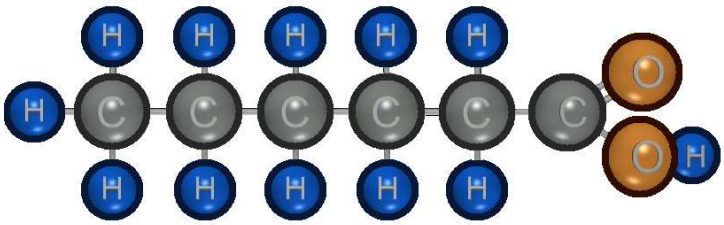


Figure 10: Long Chain Fatty Acid [35]

Because it is a matter of timing, we can craft a pulse sequence that allows us to capture the water signal while ignoring the effects of the fat signal. We use the molecules difference in relaxation time, to capture the T1 and T2 water signals. (See Figure 11) Where Tp=time between RF pulses, Tc=time after the last pulse to start the signal capture process. The pulse sequence for a

T2 Weighted Image (T2WI) uses a large time between pulses (Tp), and waits long time (Tc) before signal capture starts. As before, the RF pulse brings all the protons into phase. The signal capture process doesn't start until after the fat protons have fully relaxed, at which time the water protons have started their relaxation and are still producing a strong signal.

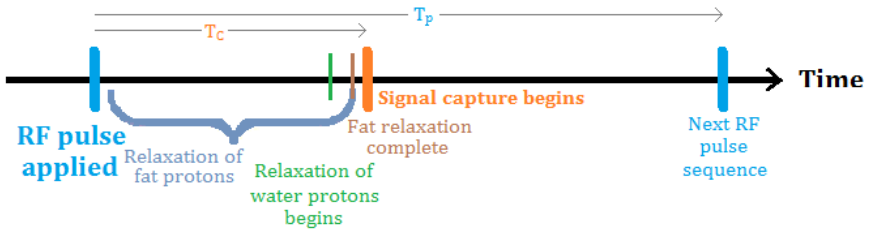


Figure 11: Pulse Sequence for a T2 Weighted Image

The pulse sequence for a T1 Weighted Image (T1WI) uses a short time between pulses (Tp), and waits a short time (Tc) before beginning the signal capture. This T1WI pulse sequence uses two magnetic pulses to complete the signal capture process. (See Figure 12)

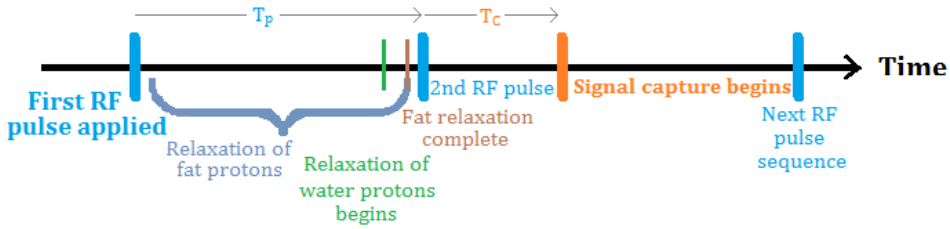


Figure 12: Pulse Sequence for a T1 Weighted Image

The T1WI process starts just like the T2WI pulse sequence. The first pulse brings all the protons into phase. The second pulse occurs after the fat protons have relaxed, while the water protons have only begun their relaxation, (the point where a T2WI would have begun the signal capture process). This causes the fat protons to return to the spinning-in-phase condition. But because the water protons are still close to their spinning-in-phase condition, the second pulse causes

more of the low-energy protons to flip into the high-energy state. Soon after the second pulse is applied, the signal capture begins. (See Figure 13) Both the fat and water protons have returned to their spinning-in-phase condition. The number of high and low energy protons in the fat molecules is balanced and in phase, thus producing a strong transverse signal. Because there are now more high-energy versus low-energy protons in the water molecules (the water protons are saturated with energy), they generate a low transvers signal as well as a small opposing longitudinal magnetic field. The tissue with a fast relaxation time (fat) produces a large signal, while the tissue with the slow relaxation time (water) produces a small signal.

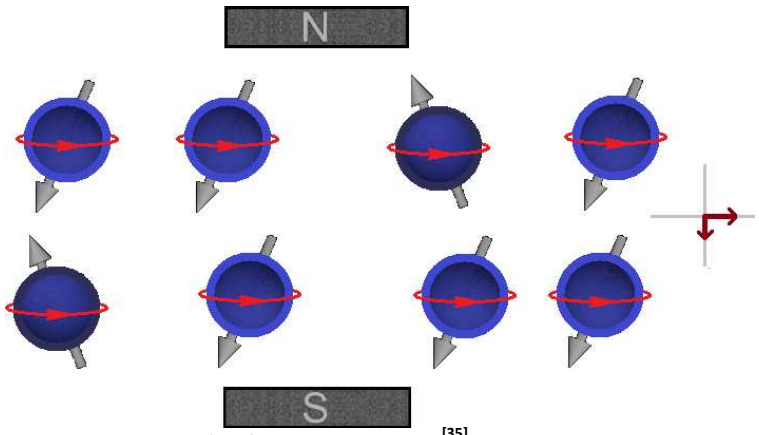


Figure 13: T1 Weighted Image Acquisition [35]

3.2 Magnetic Resonance Imaging (MRI)

A magnetic resonant image is a picture of an NMR scan. A computer monitor displays images using a 2D array of picture elements, or pixels. A typical computer screen size could be: 1024x768 pixels. A volume element, or voxel, is a 3D version of a 2D pixel. A typical MRI generates 512x512 voxels per slice. Each voxel represents a small volume of body tissue containing a small set of water molecules. The MRI machine uses NMR to generate a series of 2D voxel images. Each scanned image represents a single slice of body tissue. When these

consecutive slices are reassembled, we have a 3D voxel representation of the scanned body tissue. The water molecules in each voxel contain a number of hydrogen atoms. The MRI machine will read the scan signal from each voxel's hydrogen protons and assemble a gray scale image of the tissue. By convention, high signal intensity is displayed as white and low or no signal intensity is displayed as black.

The individual voxel signal levels are isolated and identified by creating both a phase and frequency shift of their MR signals. This is done by creating magnetic gradients. A magnetic gradient is produce along the length (head to toe) of the subject. (See Figure 14) Rather than using a fixed magnetic intensity along the Z-axis, the magnetic intensity value has a fixed sloped along the main magnetic axis. Recall that the Larmor frequency is directly related to the magnetic field strength: $f = \gamma B_0$. As a result of the sloping intensity, every slice of tissue along the gradient (main magnetic axis) has a different Larmor frequency associated with it. By selecting the appropriate resonant frequency, we can isolate a specific slice of tissue.

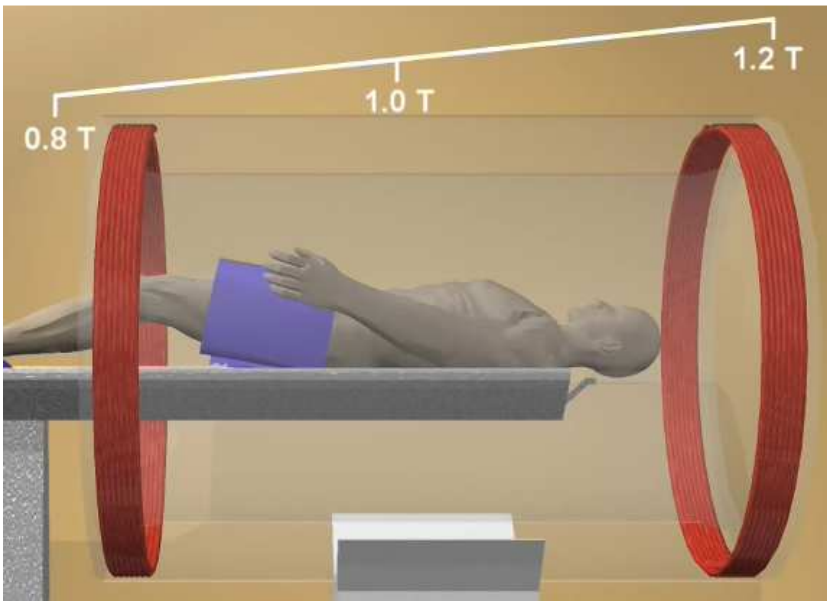


Figure 14: The red coils produce the Z-Axis Magnetic Gradient: Slice Selection [35]

By applying magnetic gradients along the other two axis of the slice, we alter the phase and frequency without altering the strength of the voxel signals. This labeling scheme allows us to encode the individual voxel signals produced by the slice of tissue. Applying a short-duration magnetic gradient along the Y axis produces the phase encoding. (See Figure 15) The spin of the magnetic moments (green arrows) in the weaker gradient region (bottom of Figure 15) slow down. While the moments in the stronger magnetic region (top of Figure 15) speed up. When the gradient is removed, the spins return to their original resonant frequency, but now they are phase shifted along the Y direction. This completes the phase encoding the Y axis.

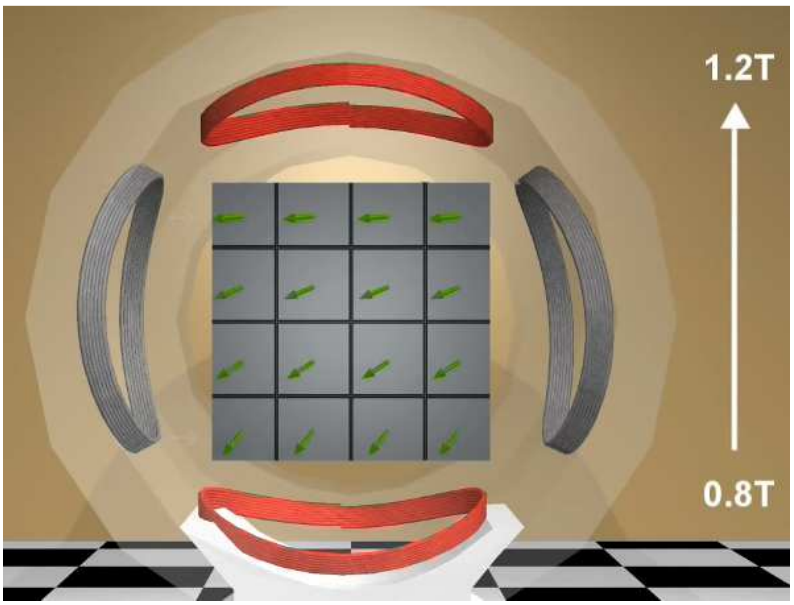


Figure 15: The red coils produce the Y-Axis Magnetic Gradient: Phase Encoding [35]

A similar operation is performed along the X axis. (See Figure 16) A magnetic gradient is applied along the X-axis (for the duration of the signal acquisition), causing the spin or signal frequency in the weaker magnetic region to slow down. The spins in the stronger magnetic region speed up, resulting in higher signal frequency. At this point in the signal capture process, every voxel location in the slice has a unique phase and frequency associated with it. This

process has essentially assigned coordinates to every voxel in the slice. By tuning the signal receiver to the appropriate phase and frequency the machine is able to scan through the entire 2D slice of tissue. The captured signal is measured and converted to a gray-scale index value. The gray scale index covers the range of white through black. The white side of the index represents a strong signal, while the black side signifies a weak signal. By stepping through this process one slice at a time, the machine is able to build a complete 3D image that shows vivid details of the tissue.

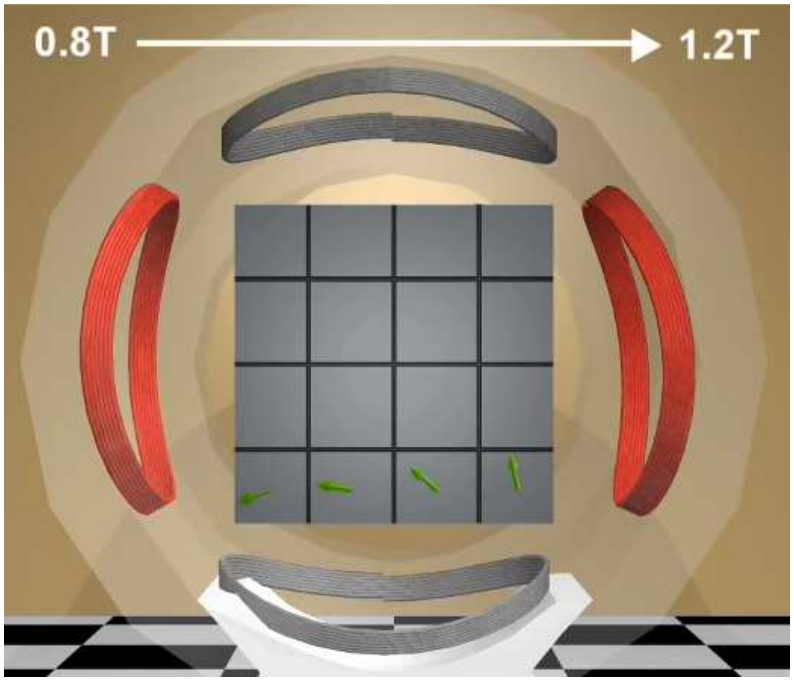


Figure 16: The red coils produce the X-Axis Magnetic Gradient: Frequency Encoding [35]

3.3 Diffusion Weighted Image (DWI)

NMR allows us to view diffusion of water *in vivo* tissue. A Diffusion Weighted Image is an NMR process that reveals the anatomic structure of the subject tissue. The image is constructed in a similar process as the above described T1 & T2 weighted images. The NMR physics are the

same: A magnetic field and RF burst are used to manipulate the magnetic poles of the hydrogen protons.

Figure 17: Random walk of a single molecule (left). Random distribution of set of molecules (right) [33]

Though it can't be seen with the naked eye, molecules in a drop of liquid water are always in constant motion. This seemingly random motion is caused by its thermal kinetic energy. This energy enables diffusion where particles tend to move from areas of high concentrations to areas of low concentration. Random refers to the particles direction of motion. (See Figure 17) Which direction the molecules move in depends on their surroundings. As a molecule performs its random walk, it interacts with other randomly moving water molecules as well as any physical restrictions. In tissue these restrictions may be a membrane, a fat molecule, or a fibrous axon. The speed of the thermal agitation is very high (~1000meters/sec)[33] due to the interaction with the surroundings, but the displacement is very small (≈20 µm in .1sec)[33]. Because the direction of the displacement is random, it is measured in terms of mean squared displacement, $\langle r^2 \rangle$. This "square" prevents negative displacements from canceling positive displacements. Einstein's equation for diffusion displacement can be expressed as: $\langle r^2 \rangle = 6D_o t$, [34] where D_o is the free diffusion coefficient of the material, and t is the time of diffusion. D_o of water at 37°C (*in vivo* body temperature) is .003mm^2/sec.[34] The random diffusion of water molecules in tissue is non-free diffusion, which is to say it is not isotropic (*i.e.* equal in all directions). Water diffusion *in vivo* is a complicated phenomenon that is affected by pressure gradients, membrane

permeability, active fluid transports (*e.g.* blood flow), and anatomical hindrance (*i.e.* fibers, cell restrictions, macromolecules, *etc.*). These many issues prevent the direct measurement of a single diffusion coefficient. Instead the term Apparent Diffusion Coefficient (ADC) is used to describe the observed diffusion *in vivo*. Any restrictions to the water molecules movement will cause the diffusion to be anisotropic, (*i.e.* diffusion varies by direction). The diffusion coefficient of a volume of tissue (*e.g.* a voxel), will be different depending on the direction from which it is observed. The average ADC for brain tissue is .0007mm^2/sec.[34] The diffusion weighted process will measure the ADC (*i.e.* mean displacement of water molecules) for every voxel, along many directions.

The diffusion weighted signal can be approximated by $S_i \cong S_0 \exp(-bADC_i)$[34], where S_i is signal intensity for a voxel in a direction *i*, S_o is the signal intensity without the magnetic diffusion gradient (basically a T2WI), b (commonly referred to as: b-value) is the diffusion sensitivity factor, and ADC_i is the apparent diffusion coefficient in direction *i*. The b is a function of the magnetic gradient amplitude G, magnetic gradient duration δ, and time between opposing magnetic gradients Δ. For short gradient times (δ), $b = \gamma^2\delta^2G^2\Delta$ [33] (γ = gyromagnetic ratio, a constant) (See Figure 18) Increasing the b increases the signal loss due to water diffusion, which increases the contrast between tissues of higher versus lower ADC values. This higher contrast comes at the cost of a lower signal to noise ratio (poor signal quality).

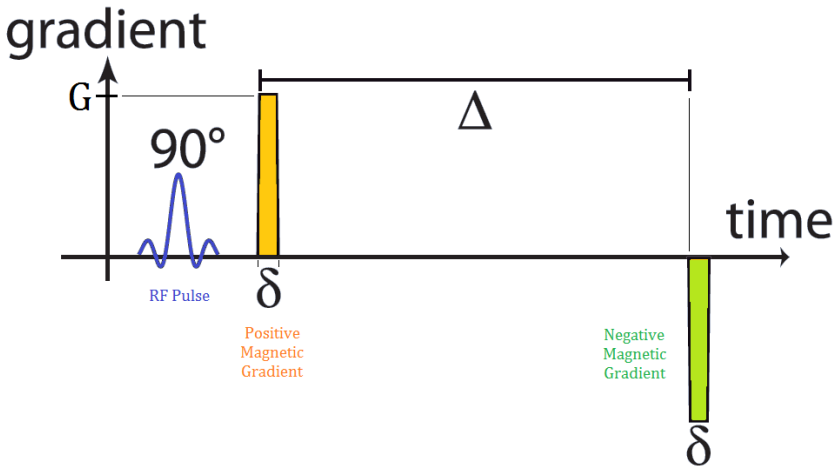


Figure 18: Pulse Sequence for a Diffusion Weighted Image [33]

The Diffusion Weighted pulse sequence begins with an RF pulse, just like the T1WI and T2WI pulse sequences. The RF energy will cause the spinning protons to phase synchronize, producing a magnetic moment, which is the source of the transverse signal. This magnetic moment is represented by the red arrows in Figure 19. The positive magnetic gradient phase (as shown in Figure 19) shifts the magnetic moments along the gradient axis. The time between gradients (Δ), is the diffusion time. The longer the diffusion time, the farther the molecules will diffuse. The negative magnetic gradient reverses the phase shift caused by the previous positive gradient, essentially rephasing the spins along the gradient axis. For those protons that have not moved during the diffusion time, the net phase shift is zero and there is no signal change ($S_i \cong S_0$). For those protons that have diffused, only partial rephasing occurs. The resulting net magnetic moment produces an S_i signal that is different from the initial no-gradient (b =0) signal, S_o. The larger the change in signal intensity, the more diffusion has occurred. This change in signal intensity is the Apparent Diffusion Coefficient that we are looking for.

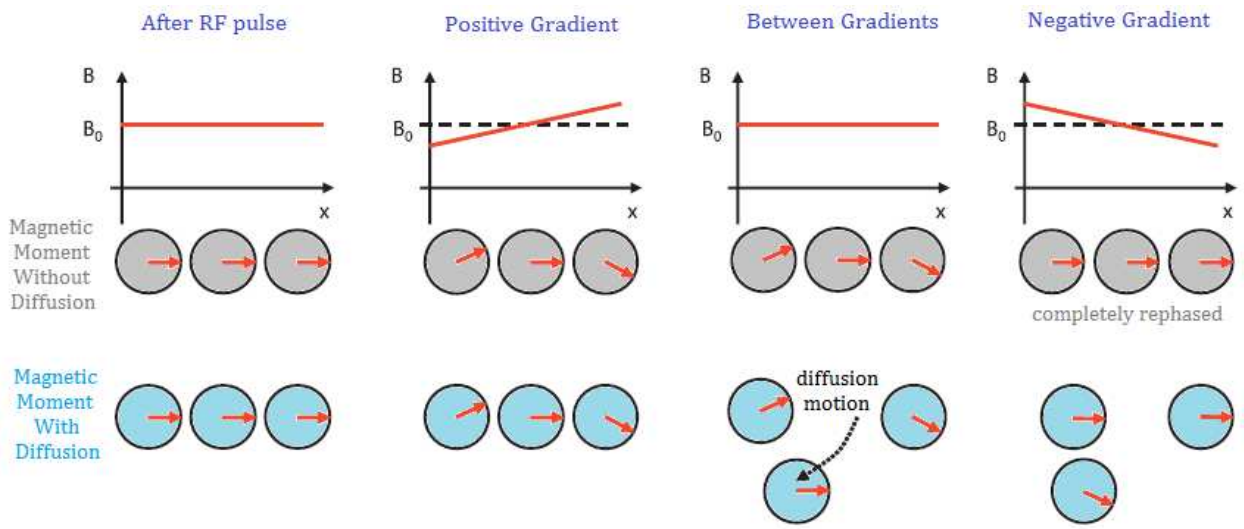


Figure 19: Diffusion Weighting Process [33], B is the magnetic field strength

Solving for the directional ADC_i in the diffusion signal equation gives us:

$ADC_i = -\ln(S_i / S_o)/b$. We are now able to convert the measured signal intensity to an Apparent Diffusion Coefficient value. By running the signal measurement and calculation on every voxel in the slice, we can assemble the resulting DWI for that magnetic gradient's direction. Each DWI shows the ADC in a single direction.

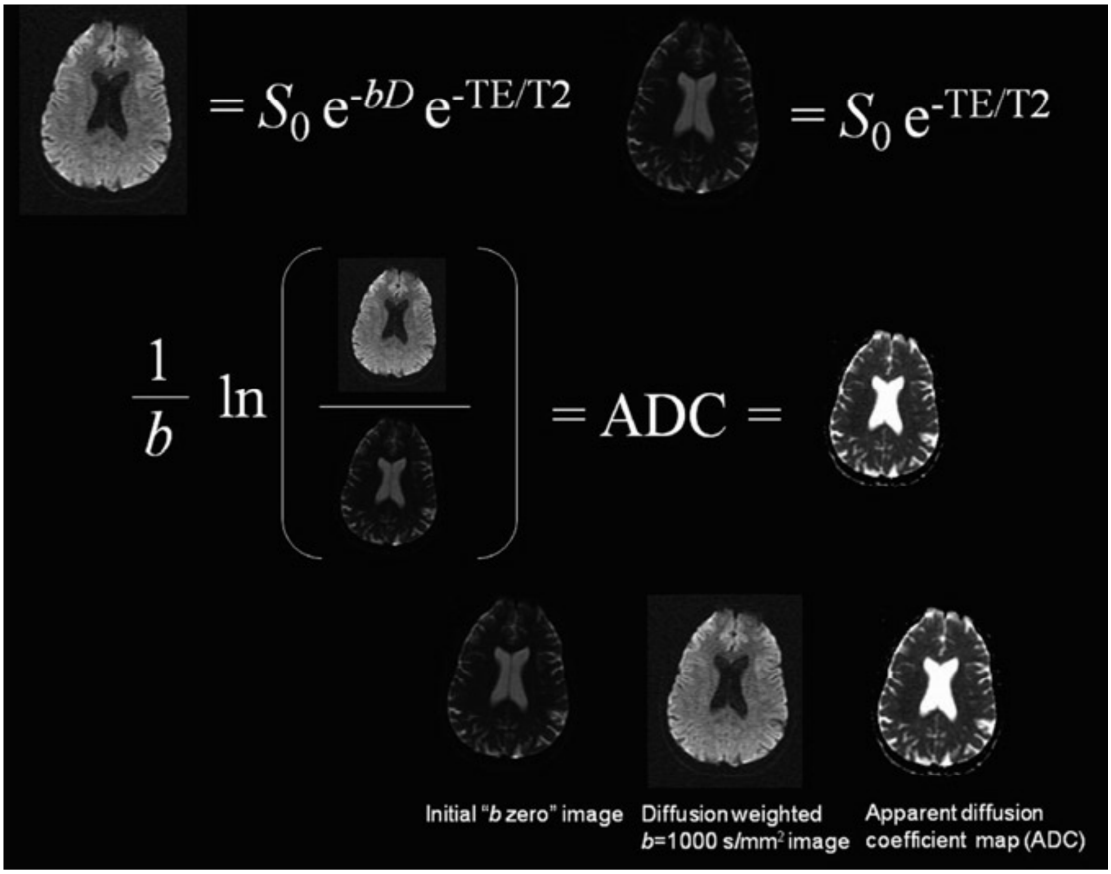


Figure 20: Converting signal intensity to ADC [34]

3.4 Diffusion Tensor Imaging (DTI)

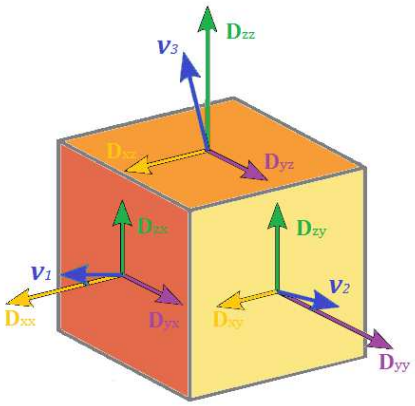
DTI allows us to see and observe the white matter tracts in the cerebrum. By combining and analyzing many non-collinear DWI scans, we are able to build a diffusion tensor image that shows the routing of neuronal fibers, ultimately providing information about the connectivity of the subject's cerebrum.

Unrestricted or free diffusion of water would show an omnidirectional diffusion coefficient. The diffusion in tissue is directionally dependent and can have many restrictions. More restrictions results in less diffusion. These heterogeneous restrictions cause the ADC to be different when observed from different directions. It is common to express quantities that change according to their spatial orientation as a tensor. A tensor is measurable and mathematically convenient. It

allows us to describe the ADC from all directions. This tensor is a 3x3 matrix that represents the ADC in 3D space.

$$\vec{\mathbf{D}} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{bmatrix}$$

Equation 1: Apparent Diffusion Coefficient Tensor for 3D space



$$v_1 = D_{xx}X + D_{yx}Y + D_{zx}Z$$
$$v_2 = D_{xy}X + D_{yy}Y + D_{zy}Z$$
$$v_3 = D_{xz}X + D_{yz}Y + D_{zz}Z$$

Figure 21: Apparent Diffusion Coefficient Tensor of a Voxel [38]

The main diagonal elements D_{xx}, D_{yy}, and D_{zz} represent the apparent diffusivity coefficients along the axes of the Cartesian coordinate system. D_{xx} is the ADC along the X axes when a field is applied along the X axes. D_{yy} is the ADC along the Y axes when a field is applied along the Y axes. D_{zz} is the ADC along the Z axes when a field is applied along the Z axes. The remaining elements represent the diffusion along orthogonal coordinate pairs. D_{xy} is the ADC along the Y axes when a field is applied along the X axes. D_{xz} is the ADC along the Z axes when a field is applied along the X axes. D_{yz} is the ADC along the Z axes when a field is applied along the Y axes. Mean squared displacement distance is a positive-only number: $<r^2>=6D_o t$. D_o along the +X axis is the same value along the –X axis direction. This means the diffusivity in the X

direction from an incident Y axis field is the same as the diffusivity in the Y direction from an incident X axes field. As a result, the ADC tensor is symmetrical. $D_{xy} = D_{yx}$ $D_{yz} = D_{zy}$ $D_{xz} = D_{zx}$ This reduces the ADC tensor to six unknowns. These six values come from seven NMR measurements: S_o and (a minimum of) six directional measurements. S_o is the base measurement, the one without a magnetic gradient. The S_{xx}, S_{xy}, S_{xz}, S_{yy}, S_{yz}, S_{zz} values are the signal intensity measurements along their respective non-collinear magnetic gradient directions. The S_{xx} value is the measured X direction signal when the magnetic gradient is applied in the X. S_{xz} is the measured Z direction signal when the magnetic gradient is applied in the X direction. Convert the signal measurements to directional ADC values, ($ADC_i = -\ln(S_i / S_o)/b$) and fill in the resulting ADC tensor value.

Water will diffuse more in the direction of least restriction. The restriction parallel to the direction of the fiber structure is generally small while it is generally much larger perpendicular to it. *i.e.* In general, the direction of diffusion is the direction of the fiber structure. The ADC tensor describes the diffusion in terms of the reference (Cartesian) coordinate system. The objective is to identify the underlying fiber structure of the tissue, which generally does not align itself with the reference coordinate system. The principal direction of the tensor is the major direction of ADC, and thus the direction of the fiber structure. To find this direction, we solve the eigenvalues of the tensor. Translate the reference coordinate system to the tensor's coordinate system. It is assumed that the eigenvector of the largest eigenvalue, is the principal direction of the tensor[37] and the direction of maximum diffusion. In the eigen equation: $\vec{A}\vec{v} = \lambda\vec{v}$, $\vec{A}$ is the square symmetrical ADC tensor, $\vec{v}$ are the eigenvectors, λ are the eigenvalues. Figure 22 demonstrates the signal scan to eigenvector conversion.

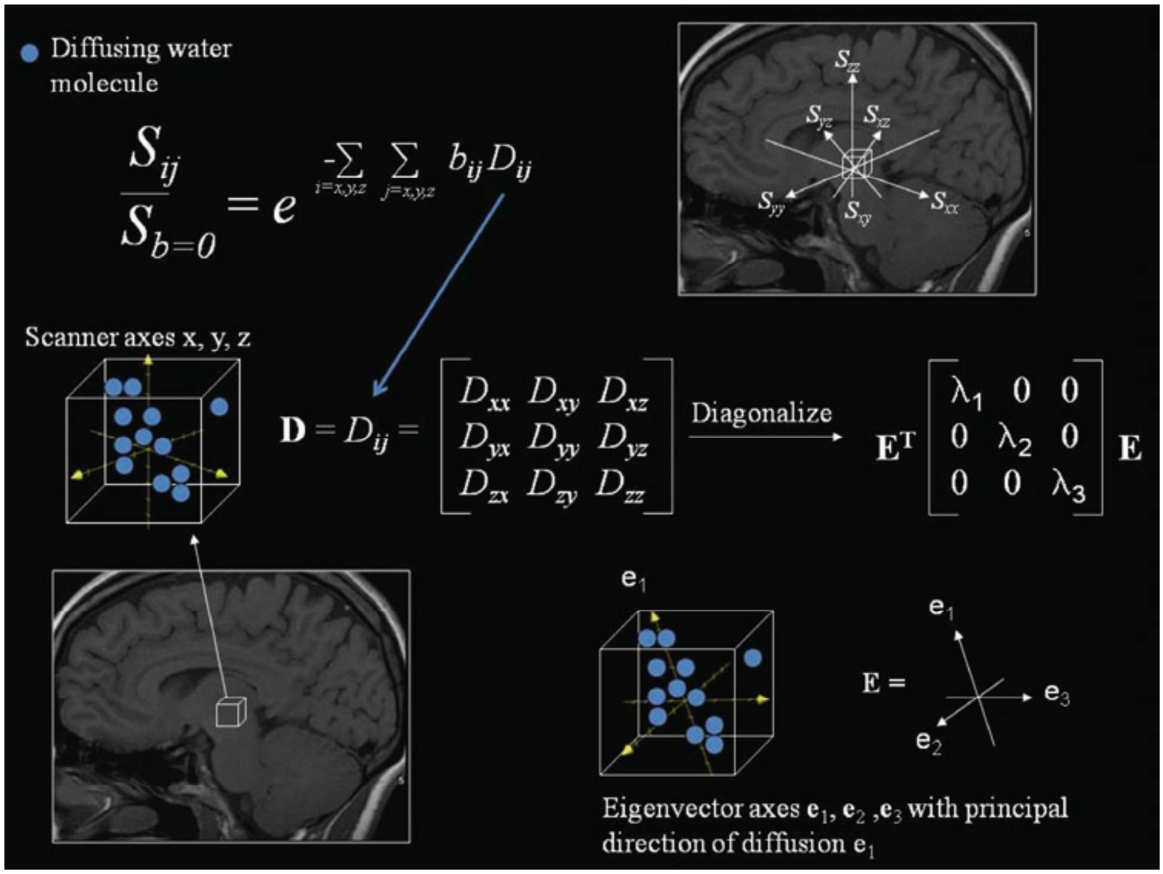


Figure 22: Converting signal intensity to eigenvector axes [34]

In unrestricted (free) diffusion the tensor is spherical: $\lambda_x = \lambda_y = \lambda_z$. (See Figure 23-left) The eigenvalues of this tensor are all approximately equal and produces no net direction. This tensor results in isotropic diffusion. In restricted diffusion the tensor is an ellipsoid. Ellipsoid tensors represent anisotropic diffusion. The longest dimension, of the ellipsoid is the presumed fiber direction. The case of a Y-axis-aligned diffusion tensor: $\lambda_y >> \lambda_x > \lambda_z$, is shown in Figure 23(right).

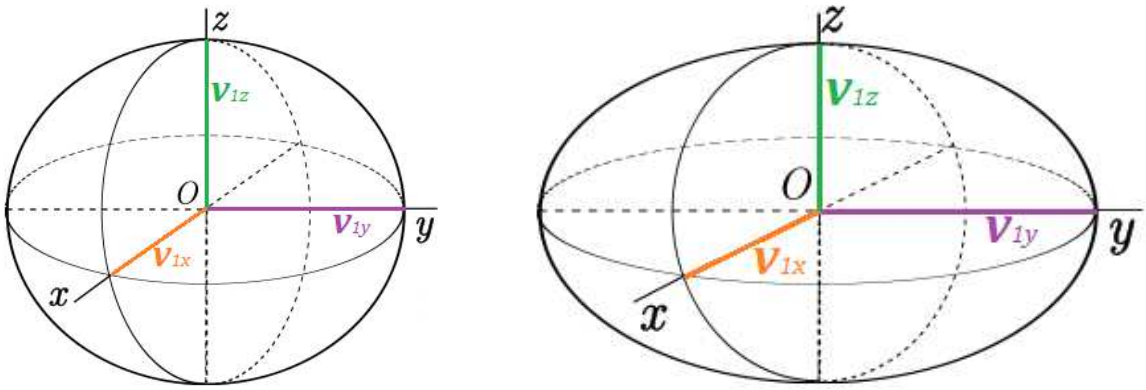


Figure 23: Isotropic Tensor (right) [39] Anisotropic Tensor (left) [40]

Cerebral Spinal Fluid (CSF) and gray matter of the brain exhibit isotropic behavior for water diffusion.[34][37] Water molecules freely diffuse in this fluid and tissue. The fibrous white matter of the cerebrum behaves in an anisotropic nature.[34][37] The exact structure causing this behavior has yet to be determined. The myelin sheath would appear to be the obvious cause, but axonal cell membranes appear to be a major cause of the water restriction.[34]

The orientation of the ellipsoid described by the tensor gives us the direction of the structure, or fibers in the white matter in that voxel. Water diffuses along the length of the fibers, and is restricted perpendicular to them. Tractography is a technique where neighboring ellipsoids are chained together revealing the fiber pathways in the cerebrum. (See Figure 24) This image is an idealized example of building fibers with tractography. Linking neighboring tensors is more difficult in the cases of crossing or kissing fibers, as it is difficult to accurately determine if fibers run adjacent or cross one another. (Diffusion tensors don't form crosses, only spheres or ellipses.) Real DTI data tends not to be as clean-cut as the example provided. A more thorough discussion on the topic can be found in [33].

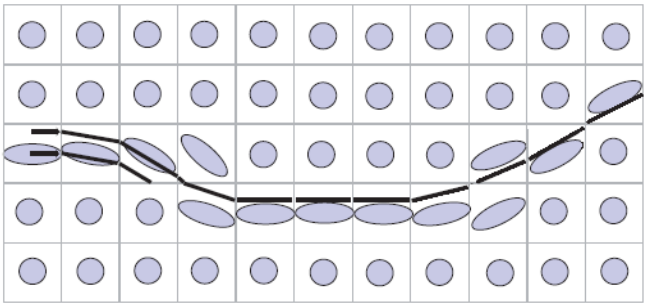


Figure 24: Chaining ellipsoids to build fiber tracts

3.5 Limitations of the process

The above description gives a theoretical overview of the principals behind building the DTI data, but there are plenty of obstacles and limitations that have not been mentioned.

- The NMR image will contain artifacts and errors that will affect the DTI data.[34]
- Diffusion is not the only source of water displacement.[34]
- A Gaussian distribution of water molecule diffusion is assumed.[34]
- NMR machines are not able to produce the narrow magnetic gradients described in the DWI pulse sequence.[33]
- There is a tradeoff between the b-value and the signals signal-to-noise-ratio (SNR).[33]
- The tensor approach to fiber tracking does not work well for the case of crossing, branching, or kissing fibers.[33][37]
- The field of tractography has not been able to determine the anatomical correctness of the tracing algorithms.[37]

The resulting DTI is not perfect, but a decent representation of the tissues anatomical structure.

There are many assumptions and approximations taken with the process that generates the final DTI data. In many cases, there are more sophisticated approaches to the processes described here, that yield better results. Further discussion and details can be found in [34], [37], [33], and [45].

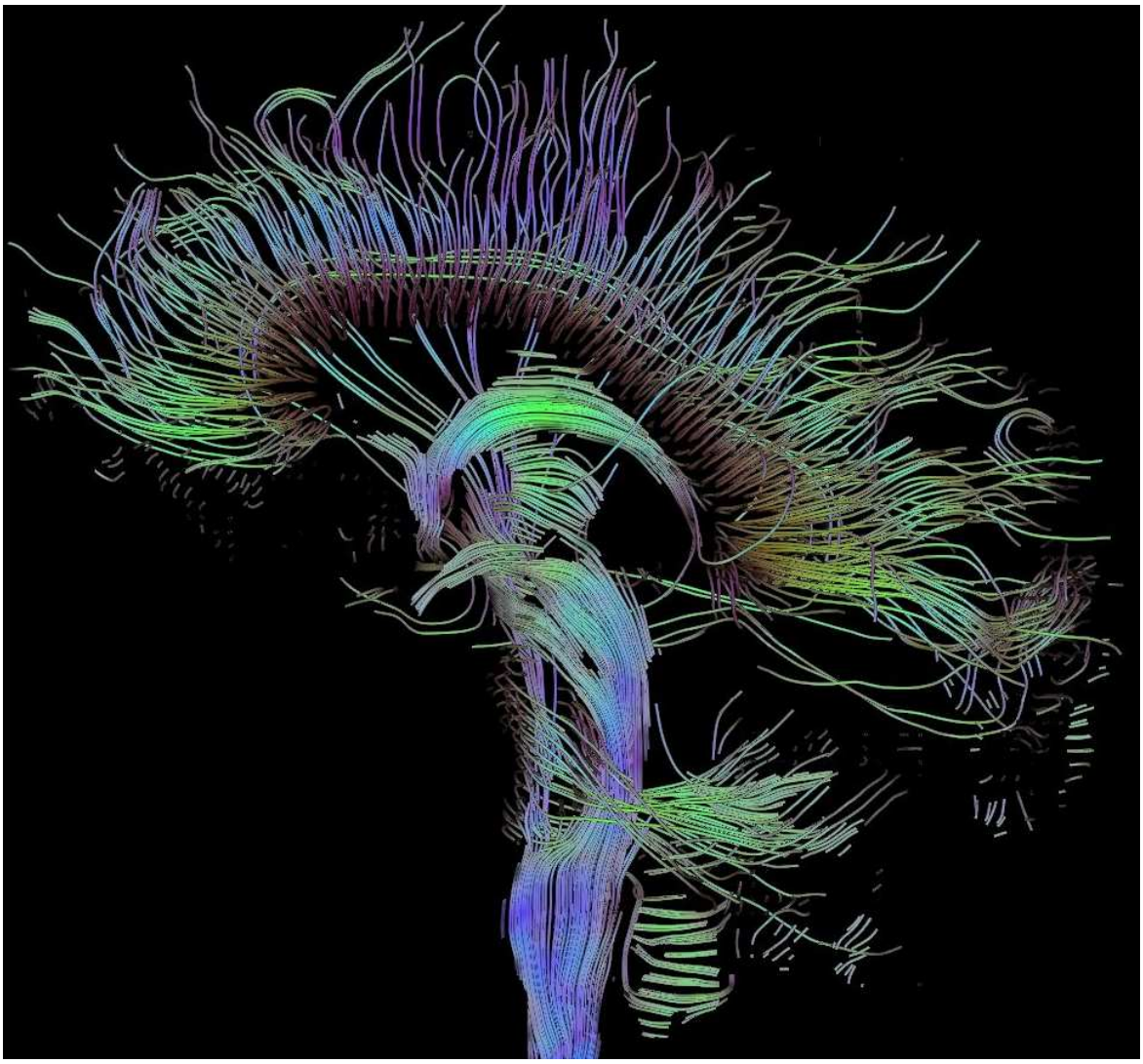


Figure 25: Diffusion Tensor Image [41] Cerebrum, Cerebellum, Medulla oblongata (Brain and Brainstem)

4. The CUDA Process

Compute Unified Device Architecture (CUDA) is a C extension language from Nvidia that performs parallel programming on their GPU devices. Many programmers are familiar with the x86 architecture where one CPU core executes one thread at a time. The single core will leverage the chips resources to quickly execute the single thread's task. If there is a list of 3000 numbers that needed to be added to another list of 3000 numbers, the CPU cache would load-up the data and the core would read the data, add the data, and write the results, one number/item at a time. This approach is referred to Single Instruction Single Data (SISD). The GPU uses a scalable Single Instruction Multiple Data approach (SIMD). To perform the same 3000 number add routine, the data is loaded to the GPU, from which the numbers are read, added, and results written one batch at a time. Where the CPU will add the numbers one at a time, the GPU will add the numbers one batch at a time. In the time it takes the CPU to serially add 2 numbers, the GPU is adding 64 numbers in a parallel approach. The difference in performance can be dramatic. Efficiency improvements are expressed in terms of speedup factors. A speedup factor of 2 means the task's runtime has been cut in half. What took 10 minutes, now only takes 5minutes. It is not uncommon to see speedup factors of 10 or more. It depends on the task and what resources are available to perform it.

4.1 Basic CPU-GPU architecture

The CPU, being the main processor, can support many GPUs, the coprocessors. (See Figure 26) The CPU (called the host), assigns tasks (called kernels), to the GPUs (called devices). Once received, the GPU will execute the task autonomously while the CPU can either wait for a GPU response or continue its serial execution.

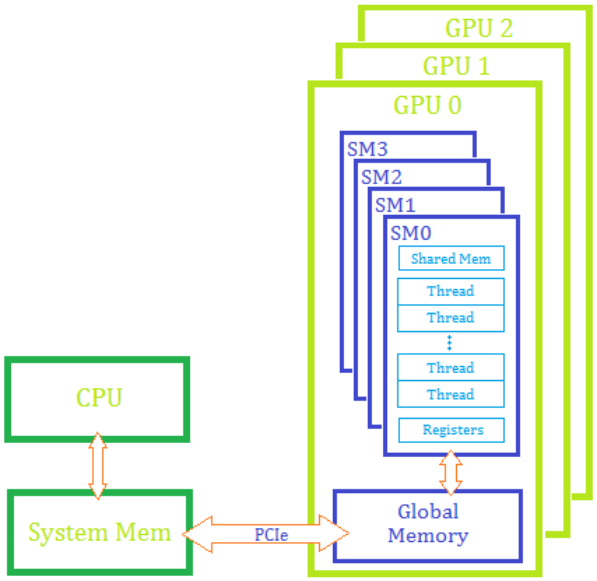


Figure 26: Single CPU processor and the many GPU coprocessors

Nvidia's basic work unit is the Streaming Multiprocessor (SM), the equivalent to a CPU core. The SM is capable of supporting the execution of thousands of co-resident threads. It has shared memory, a set of registers, and a set of processors. A single clock cycle of the SM will cause the execution of a same-instruction on a batch of concurrent threads, referred to as a warp. Outside of the SMs is a heap of device memory called Global Memory (GM). (See Figure 26) This memory is accessible by the CPU by means of the PCIe bus and the array of SMs.

4.2 GPU memory architecture

The GPU uses many memory formats. They are distributed throughout the device to reduce the execution time. From slowest to fastest access times they are: global, constant, shared, and register.

- Global memory is the pathway for getting data in and out of the GPU. It is separate from system (host) memory space, and is not directly visible by the host. It is located outside the GPU processor which is why it is the slowest of the GPU memories. It is accessible by the whole GPU device.

- Constant memory, as its name implies is fast read-only memory. It is located outside the GPU processor. It is cached and visible by any thread, in any SM.
- Shared memory is on-chip and is located near the SMs. This memory is accessible by any thread in the SM. It is local to that SM and not visible by any other SM. This memory is a means of getting the data close to the processor that will use it. It is called shared memory because it is shared with any thread in that SM. Using shared memory is an easy way of passing results between threads in the SM.
- Registers are the fastest memory storage format. They are on-chip and are accessible by a single thread. They are only valid for the lifetime of that thread.

4.2.1 Coalescing global memory

Global memory access can be faster or slower, depending on how it is accessed. The read cycle for global memory has been optimized to feed a complete warp. If, for example, the GPU routine needs to read some data in from memory, this would be the same as saying each thread in the SM is calling for a 4 byte word from global memory. To satisfy a full warp of 32 threads, the device needs to read 4*32=128 bytes of global memory. The optimized hardware is capable of reading all 128 bytes in a single read cycle. (See Figure 27) If the data has been stored properly, the data request could be satisfied with one read cycle. This is referred to as data coalescing. If the data in global memory has been stored very poorly (*i.e.* non-coalesced), it may require 32 read cycles to get enough data to satisfy the full warp. This will have a big impact on the kernel's runtime. An efficient program would opt for a single read, rather than the many read cycles. How the data is stored in memory and how it is accessed is governed by the software author.

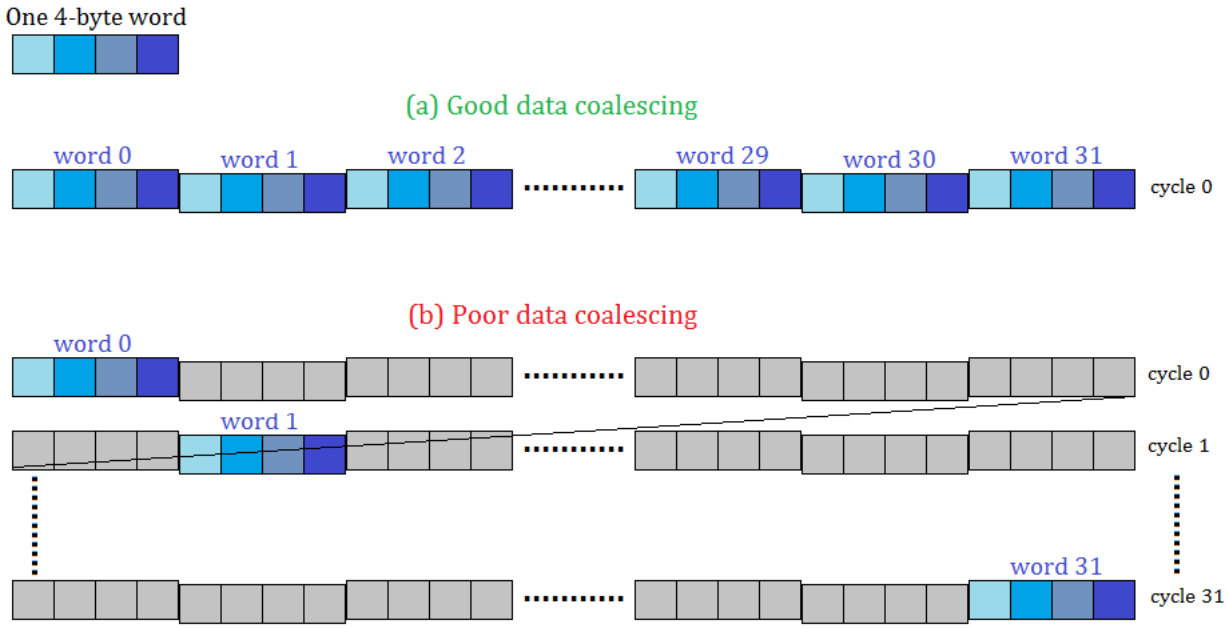


Figure 27: Global memory coalescing a) Single cycle access b) Many cycle access

4.2.2 Shared memory bank conflicts

In a similar fashion, shared memory can be faster or slower depending on how it is accessed. Shared memory is arranged as banks of striped memory. (See Figure 28) This storage format allows parallel access to the data. The memory bus allows each thread in the warp to have access to any bank. If the parallel code requests a data word, and each requested data word is in its own bank, a single read cycle will satisfy the data request for the whole warp. If the data is not equally distributed to the banks, a bank conflict results and more than one data fetch cycle will be needed to satisfy the request. For example, in Figure 28, if *thread0* is requesting *word0*, (the first word of *bank0*), and *thread1* is requesting *word32*, (the second word of the same *bank0*), the memory bus for *bank0* can only accommodate one word at a time. A bank conflict exists and the code will need to issue two read cycles to satisfy the data request. The code will still execute properly, but it will take longer to run. The kernel would be more efficient if the data words were better distributed. Bank conflicts are caused or resolved by the software author.

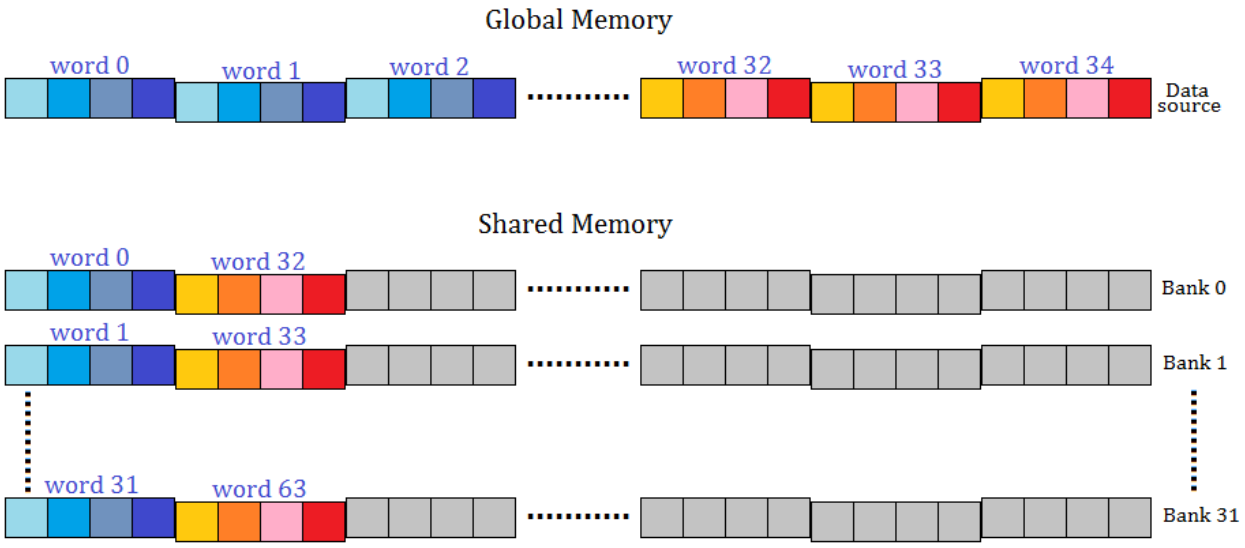


Figure 28: Striped Distribution of Shared Memory

The shared memory bus also supports a broadcast function where one word from one bank can be broadcasted to all threads in the warp, in a single read cycle.

4.2.3 Race conditions

With many processors having access to the same memory, there is plenty of opportunity for data hazards. The most obvious would be many threads writing to the same memory location. This is a rather common issue. A memory location can only hold one value at a time. If each thread writes a different value, how can we deterministically control which value gets written? This can accomplished with the Atomic functions. These functions only allow one thread to access the memory location at a time. This comes at the cost of "pausing" all other threads that may be trying to access the memory location. Use of Atomic functions can severely affect the efficiency of the kernel. It is generally better to take a different design approach and avoid the use of Atomic functions.

Data hazards occur when data is modified at different stages of execution. For example, a Write After Read (WAR) data hazard occurs when a value is read before it has been written by a prior modifying stage. A dependency exists between the write stage and the read stage. For example,

if *thread22* reads a value in memory, but that value must first be written by *thread52* of the same kernel, a WAR data hazard may exist. A tool that prevents data hazards inside a kernel is *__syncthreads()*. This statement prevents further execution of the kernel until all active threads have reached this command and all reads or writes to shared or global memory have been completed. As the name implies, it is a thread synchronization point in the code. Execution will be stalled at the sync-point until all active threads in the kernel have reached this common point.

4.3 GPU programming model

The basic hardware (HW) work unit in the GPU is the streaming multiprocessor. This SM has a batch of logic-units, each of which concurrently executes the same kernel code. Each logic-unit in the batch will perform the same operation in the same machine cycle, as all other parallel logic-units. Expressed another way, all the logic-units execute exactly the same kernel code, at exactly the same time. The number of parallel logic-units in the SM is fixed. If the size of the task is larger than the number of logic-units (warp size) in the SM, the task is split into smaller pieces, where each piece of the task will get runtime on the logic-units. Essentially the scalable task is being run, one warp of threads at a time.

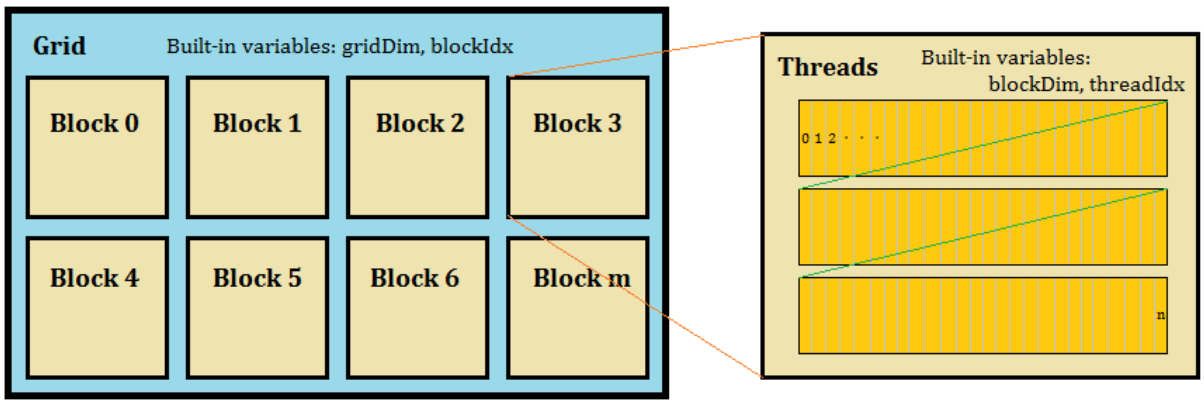


Figure 29: Grid is a group of blocks Block is a group of threads

The scalable software model works with grids, blocks and threads. (See Figure 29) A grid contains a number of blocks, and a block contains a number of threads. A thread represents a single instance of the task. From the earlier example, a task that adds two sets of 3000 numbers, we could breakdown the task into 3000 threads. Each thread would read a number from each data-set, add them, and save the result. If the GPU we are using has maximum block size of 1024 threads, we would need at least *ceil*(3000/1024)= 3 blocks to perform the task. One could scale the task to 1 grid of 3 blocks, each with 1000 threads. Three grids of 1 block, each with 1000 threads would also work. One grid of 6 blocks, each with 500 threads would also work. Scalability allows us to breakdown the task into whatever size the problem requires. How the task is breakdown depends on what resources are available, and how quickly we want the task to run.

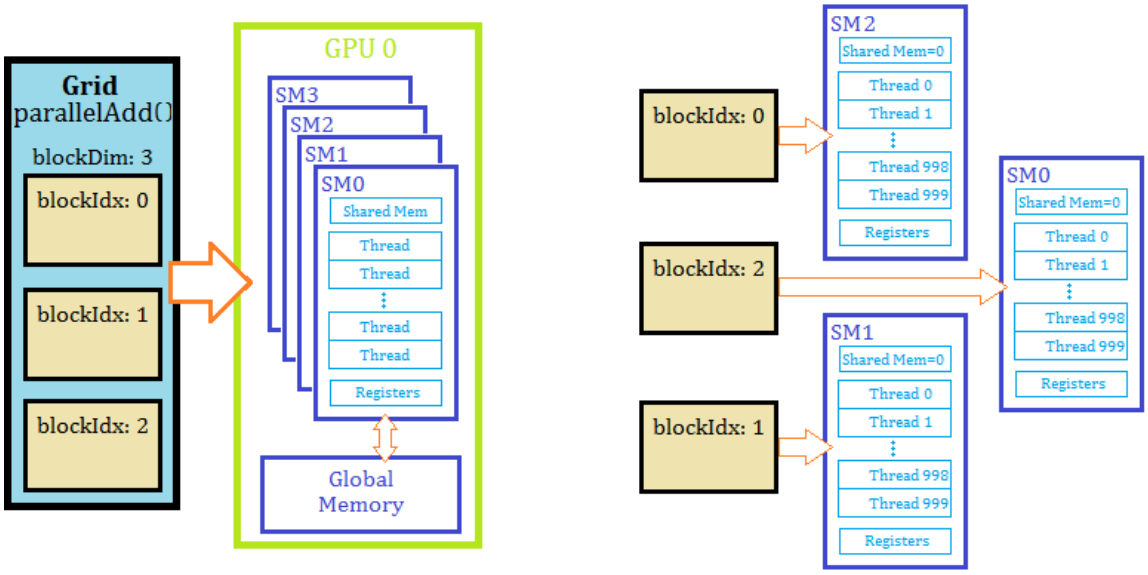


Figure 30: Block to Streaming Multiprocessor Assignment

The blocks in a grid are assigned to an SM for autonomous execution. (See Figure 30) The threads in the block are executed in batches of warp. If the warp size is 32, the SM will be executing 32 threads concurrently. Each block's execution runs independent of other blocks.

The execution and shared memory in one block are not directly available to other blocks. Blocks asynchronously execute kernel code in the sub-environment of the SM. Passing data (sharing results) between blocks is done through global memory. Asynchronously refers to the order of execution. There is no guaranteed order of execution of the blocks or the threads. Any block in the grid may be the first block to execute, as any warp in the block may be the first warp to execute. The hardware supports very fast context switching. If the code hits a sync point, the current warp will stall and another warp will load and begin execution. While the warp is executing, every thread in that warp is executing the exact same code. Note that warp execution is concurrent, and expect block execution to be nonconsecutive.

The examples in Code block 2 and Code block 3 perform the same operation of adding two sets of 3000 numbers. Code block 2 uses only the CPU, while Code block 3 uses both the CPU and the GPU. There is nothing particular interesting about the CPU-only approach in Code block 2. It is a single loop that adds two numbers and saves the result.

```
// Compile w/: nvcc
#define numCount  3000          // size of data set
int main(void) {
    int dataSetA[numCount], dataSetB[numCount];     // the numbers
    to add
    int dataResult[numCount];     // the add result
    for (int index=0;  index < numCount; index++)
       dataResult[index]= dataSetA[index] + dataSetB[index];
    return 0;
}
```

Code block 2: CPU code for adding 3000 number

As with the CPU approach, the GPU routine (see Code block 3) starts with the datasets in system memory. The SM's in the GPU can not access CPU system memory. The datasets must be copied from CPU system memory to the GPU global memory. To do this, we first create a few host pointers that will hold the device addresses of the datasets in global memory. Next we

reserve the necessary chunks of global memory and set the pointers accordingly, using *cudaMalloc()* calls. *cudaMalloc()* reserves device global memory in the same fashion as the familiar *malloc()* does for reserving CPU system memory. *cudaMemcpy()* will copy the datasets from CPU system memory to the GPU global memory. *cudaMemcpy()* performs the same operation as the familiar *memcpy()* function with an additional parameter which specifies the direction of the copy operation. (*cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice*)

Now that the data is in the GPU, we can launch the *parallelAdd()* kernel. The format of the launch command is: *kernelName<<<resources>>>(parameters).* Aside from the *<<<resources>>>* portion of the call it works the same as a function call for the CPU system. *<<<resources>>>* tells the GPU scheduler what GPU resources the kernel needs for execution. The format is *<<<nBlocks, nThreads, nSharedMem, streamId>>>*. The first two items are a description of the grid. *nBlocks* is the number of blocks in the grid assigned to *gridDim* (a built-in CUDA variable). *nThreads* is the number of threads in each block assigned to *blockDim* (a built-in CUDA variable). *nSharedMem* is the amount of shared memory, in bytes, each block will use. *streamId* is the identifier of the stream in which the kernel will run in. The *streamId* is a method for running many synchronous processes, asynchronously. The identifier is supplied by the GPU scheduler and is used to ensure the synchronous execution of many parallel running processes. These processes can be similar or completely different from each other. In this case, the identifier is 0, signifying the default stream. Only the *stream0* process runs synchronously with the CPU. This means CPU execution will pause until the GPU process completes. This keeps the CPU and GPU execution synchronized. GPU execution still runs asynchronously with other streams inside the GPU, but the CPU execution does not continue until the *stream0* GPU

process completes. All other nonzero *streamId*'s cause those processes to run asynchronously with the CPU. The CPU launces GPU kernel(s) and then proceeds with further system execution, not waiting on the GPU.

When CPU execution resumes, the parallelAdd() kernel has finished. The results are copied from the GPU memory-space to the CPU system memory-space, (with *cudaMemcpy()*). The add operation is now complete. All that remains is to return the reserved global memory back to the GPU heap, (with *cudaFree()*).

The parallelAdd() kernel in Code block 3 is straight forward. (The grayed lines are optional debug code.) Each thread will read two numbers from global memory, add them together, and write the result back to global memory. Each thread needs to know which dataset elements it will operate on. This index is calculated using the built-in CUDA variables. Each thread and each block has a unique identifier, *threadIdx* and *blockIdx* respectively. The number of blocks in the grid can be read with the *gridDim* variable. The number of threads in any block can be read with the *blockDim* variable.

The work done by the kernel is basically a single line of code that undergoes parallel execution. The *IF* statement protects the kernel from execution and memory overrun. Instead of 3000 numbers, maybe the dataset has *count*=2999 numbers. If we ran this kernel without the *IF* test, the GPU would launch *gridDim***blockDim*=3*1000=3000 threads. The last thread would be unnecessary to complete the work, but it will modify global memory. Memory that may be outside the kernels reserved memory space, potentially damaging the data from some other kernel. These types of bugs can be very difficult to track down. Though it isn't necessary for this example, it's good programing practice to use this approach.

```
// Compile w/: nvcc -arch=sm_20
#define numCount  3000          // size of data set

// The GPU is the Device, the coprocessor
__global__ void parallelAdd(int count, int* dataA, int*dataB, int* sumRslt) {
    if( (threadIdx.x == 0) && (blockIdx.x ==0) )  // print only once
       printf("Running parallelAdd() using %d blocks, %d threads\n", gridDim.x, blockDim.x);
    int dataIdx= blockIdx.x*blockDim.x + threadIdx.x;      // date element to work on
    printf("Hello from device: block %d, thread %d\n", blockIdx.x, threadIdx.x);   // every
    thread will print this
    if(dataIdx < count)
       sumRslt[dataIdx]= dataA[dataIdx] + dataB[dataIdx];     // all the work done here
}

// The CPU is the Host, the main or lead processor
__host__ int main(void) {
    int dataSetA[numCount], dataSetB[numCount];    // the numbers to add
    int dataResult[numCount];                          // the results
    int* numA= NULL;          // CPU memory that
    int* numB= NULL;          // holds a pointer to
    int* numRslt= NULL;       // GPU global memory
    printf("Hello from the CPU\n");
    cudaMalloc(&numA, sizeof(int)*numCount);       // grab some device global memory
    cudaMalloc(&numB, sizeof(int)*numCount);
    cudaMalloc(&numRslt, sizeof(int)*numCount);
       // copy the data from CPU system memory to the GPU global memory
    cudaMemcpy(numA, dataSetA, sizeof(int)*numCount, cudaMemcpyHostToDevice);
    cudaMemcpy(numB, dataSetB, sizeof(int)*numCount, cudaMemcpyHostToDevice);
    parallelAdd<<<3,1000,0,0>>>(numCount, numA, numB, numRslt);     // launch the kernel
    cudaMemcpy(dataResult, numRslt, sizeof(int)*numCount, cudaMemcpyDeviceToHost);  //
    get results
    cudaFree(numRslt);     // give back the global memory
    cudaFree(numB);
    cudaFree(numA);
    return 0;
}
```

Code block 3: CPU & GPU code for adding 3000 numbers

The *__host__* and *__global__* terms are compiler qualifiers. The *__host__* qualifier is optional as any function without a qualifier defaults to a host function. As the name implies, these functions are compiled for execution by the host. The *__global__* qualifier signifies a kernel function. This code is compiled for device execution. Kernel functions have no return value.

They must return a *void*. Another qualifier not used in this example is *__device__*, which also designates code for device execution. These *__device__* functions are subroutine code that can only be called from kernels. A function can have more than one qualifier. A function with *__global__* and *__host__* qualifiers would tell the compilers this code will be executed on both the host and device. A host version as well as a device version of the code will be built.

Error handling is done though status words. All CUDA APIs (host code) return a status word, such as *cudaSuccess*. It is good practice to check all returned status words. This number can be converted to an ASCII string with a call to *cudaGetErrorString()*. Kernel calls have no return values. Their status can be checked by retrieving the last recorded error, *cudaGetLastError()*. As with the API status, a status word/number is returned. As the function call implies, only the last detected runtime error of the same host thread is returned. The returned status word may be an error from the last or an earlier runtime call. Errors are noted, successes are not.

The GPU version has many more lines of code than the CPU version. One might think the CPU version would run faster, and it would if the dataset is only a few thousand elements. The cache capabilities of the CPU and the small dataset, allow the CPU-only approach to outperform the CPU-GPU pair. If instead the dataset had 30K data elements, the parallel execution of the CPU-GPU pair would outperform the CPU-only execution. As the dataset gets larger the performance difference between the CPU-only and the CPU-GPU pair gets larger. A CPU is a one-size-fits-all approach to problem solving. The strength of the GPU is its ability to quickly perform the same operation on many pieces of data.

5. CudaDoTraceMap Process

The most time consuming routine in the DICCCOL process is the predictROI2 routine. The predictROI2 process (see Code block 1) finds the best match for each of the 358 cortical landmarks. A set of subject bundles is selected in the region of interest (ROI) surrounding each landmark. Each of these bundles will be processed by the CudaDoTraceMap process. This process will examine all the fibers in the bundle and compare the fiber orientation against the preprocessed reference library results. Each fiber in the bundle consists of a series of fiber points. These fiber points are broken down into segments. These segments are converted to trace (orientation) points. These trace points are then converted to a feature count. This feature count is then compared against the feature count results of the reference library. At the end of the process a match score is returned. The bundle with the closest match to the reference library is chosen as the best match of the ROI.

The parallel processing approach is performed by the CudaDoTraceMap process. (See Code block 4) It comprises three processes. These processes are where predictROI2 spends most of its time. Any small reductions in these processes results in large runtime savings.

```
CudaDoTraceMap() {
    for(fiberIdx=0;  fiberIdx<nFibers;
    fiberIdx++) {
        GpuBundleToCovariance();
        Gpu3CovarianceToPrincipalDir();
        GpuPrincipalDirToFeatureScore();
    }
}
```

Code block 4: Pseudo code of main CUDA process

5.1 Bundle To Covariance

This process will reduce the input bundle of fibers to a set of 3 dimensional covariance matrices. Each matrix represents a single segment of fiber. A segment contains 16 fiber-points of data. Successive segments of fiber overlap each other by half. (See Figure 32) Each of the 16 fiber-points in the segment contains the 3 dimensional fiber-point data that will generate the 3x3 covariance matrix for that segment, which is the first step in the Principal Component Analysis[2] (PCA) of the segment.

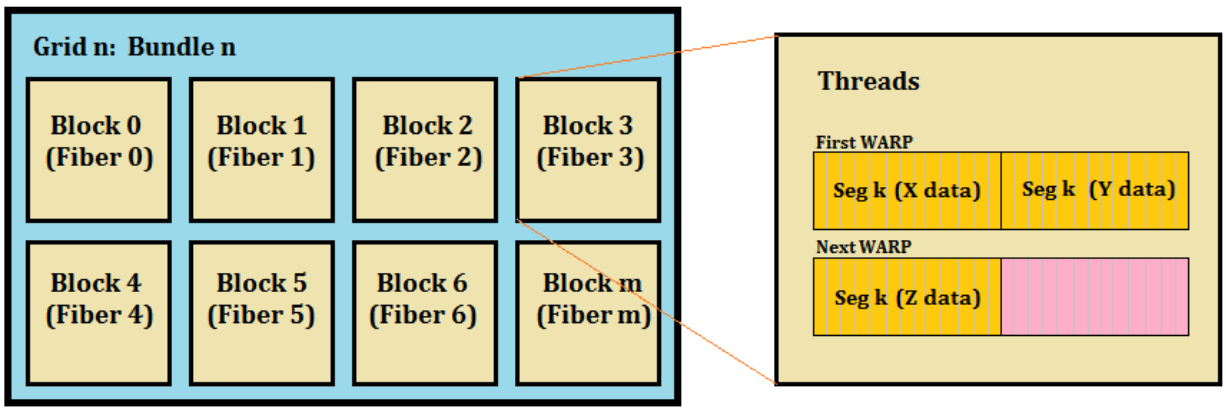


Figure 31: Bundle to Covariance Grid Breakdown (Orange) used threads (Pink) unused threads

A single GPU grid will process a single bundle of fibers. (See Figure 31) Each bundle contains a variable number of fibers, but is known to be much less than the GPU's maximum grid dimension. Each block in the grid will process a single fiber. Each fiber contains a variable number of segments. The number of segments is known to be much less than the GPU's maximum block dimension. Each thread in the block processes a single (XYZ) coordinate of the segments. Each thread represents an X, Y, or Z data point. Each pass of this process converts a single segment of fiber data to a single 3x3 covariance matrix.

```
// the kernel call (from the CPU)
GpuBundleToCovariance <<<numFibersInBundle, 3*segmentSize,..>>> ();
```

```
// one bundle per grid, one fiber per block, each thread works on a single
segmentPt
GpuBundleToCovariance() {
    for(segmentIdx=0;  segmentIdx<segmentsInFiber;  segmentIdx++) {
        Calculate dimension average: (x,y,z)
        Calculate dimension difference: segmentPt - dimension average (x,y,z)
        Calculate dimension variance: (XX, YY, ZZ)
        Calculate covariance: (XY, XZ, YZ)
        Calculate segment general direction: (last segmentPt - first segmentPt)
    }
}
```

Code block 5: Pseudo code of the GpuBundleToCovariance process

The thread-block processes the fiber one segment at a time. (See Code block 5) The 16 point segment size allows the covariance calculation to be completed by two warps. (WarpSize= 32 threads) Though it would have been a more direct approach to use 3 warps, I found it was faster to split the coordinates among half-warps. This means the covariance calculation (Equation 2) occurs in sizes of half-warp. Each half-warp uses a reduction technique to calculate the X,Y,Z coordinate of the variance (XX, YY, ZZ) values. The covariance values (XY, YX, YZ, ZY, XZ, ZX) are then calculated and saved for the next processing step. The covariance values are symmetrical: XY=YX, YZ=ZY, XZ=ZX

$$\mathrm{cov}_{IJ} = \frac{\sum_{i=1}^{n} \left(I_i - \bar{I}\right)\left(J_i - \bar{J}\right)}{n}$$

Equation 2: Covariance calculation I,J=dimension ID, n=segment length

A single bundle of 95 fibers with 11,104 fiber-points will be converted to 1,293 single precision covariance matrices in approximately 175uSec.

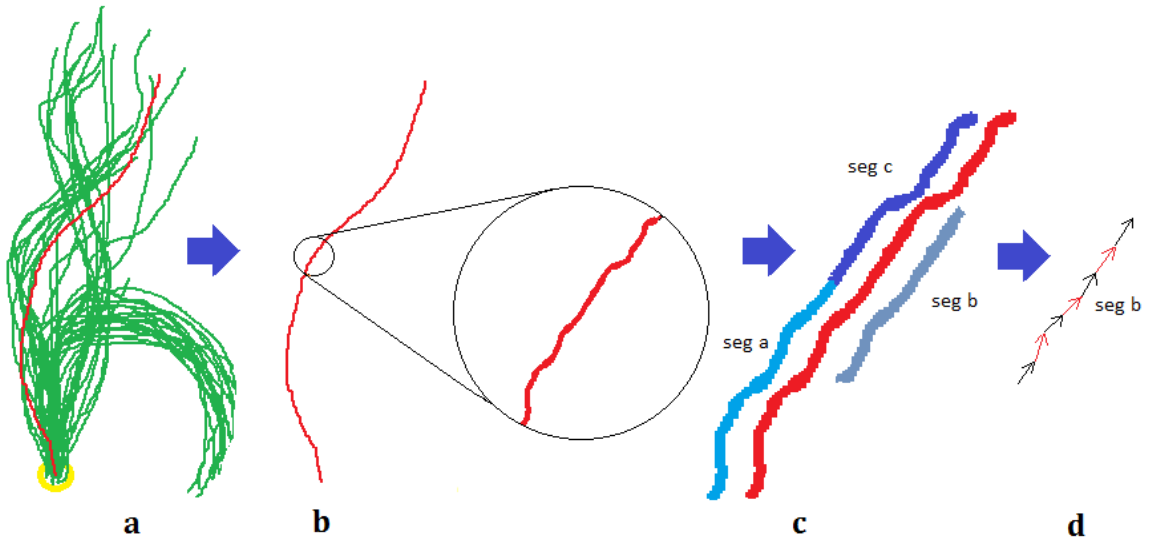


Figure 32: Fiber Segmentation (a) Bundle of fibers (b) Single fiber (c) Fiber Segmentation (d) Single Segment Fiber Points

5.2 Covariance Matrix To Principal Direction Vector

This process will convert the covariance data to a Principal Direction vector, the major PCA vector. This is done by solving the major eigenvector of the covariance matrix. Most eigen solvers are designed for hundreds of dimensions and high accuracy. None of which are required here. The 3D matrix is symmetrical about the main diagonal (variance values: XX, YY, ZZ). The final stage of the TraceMap process only requires a single digit of precision, and only single precision numbers are needed. These issues are advantages and help us quickly solve this eigen problem.

```
// the kernel call (from the CPU)
Gpu3CovarianceToPrincipalDir <<<1, sizeOfCovarMatrix,..>>> ();

// one bundle per grid, one block, each thread works on a single matrix
element
Gpu3CovarianceToPrincipalDir() {
    for(segmentIdx=0;  segmentIdx<segmentsInBundle;  segmentIdx+=3)
    {
        Calculate principal direction value for 3 segments
        Calculate principal direction vector for 3 segments
```

```
        }
    }
```

Code block 6: Pseudo code of the Gpu3CovarianceToPrincipalDir process

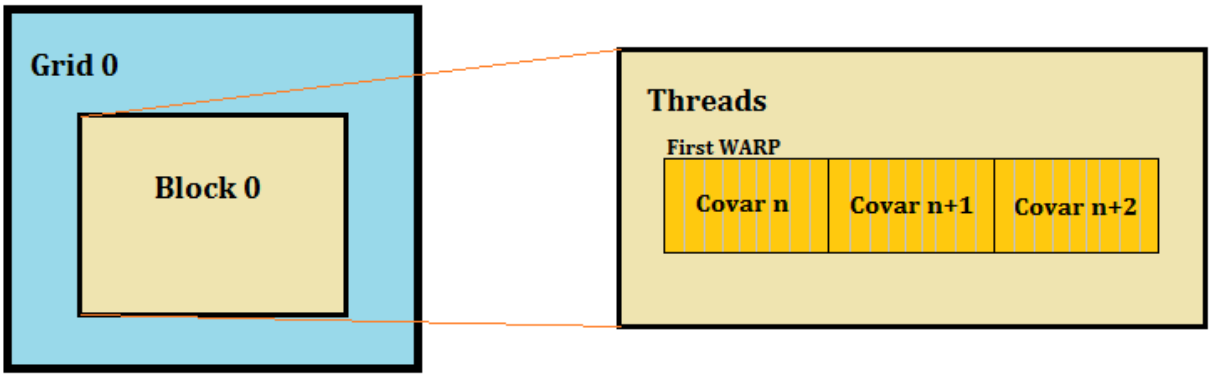


Figure 33: Covariance to Principal Direction Grid Breakdown

A single-block grid will calculate every Principal Direction vector in the bundle. (See Code block 6, Figure 33, and Figure 35a) Each thread in the block represents a single value of the covariance matrix. Each pass of this process calculates the Principal Direction vector for three covariance matrices.

The principal-direction-vector is the major eigenvector of the covariance matrix. To find the eigenvector (**v**) we first need to find the major eigenvalue (λ) of the covariance matrix (**A**). **A** x **v**= λ **v** (See Figure 34)

Step 1: Calculate the deviatoric matrix: **A'**= **A** – 1/3 **I A**

Now **A'** x **v**= η **v**, where $\eta_i = \lambda_i - 1/3\ tr\mathbf{A}$. $tr\mathbf{A'}=0$, $\eta^3 - j_2 \eta - j_3 = 0$

Step 2: Solve the coefficients j_2 and j_3: $j_2 = \frac{1}{2}tr(\mathbf{A'A'})$, $j_3 = det\mathbf{A'}$

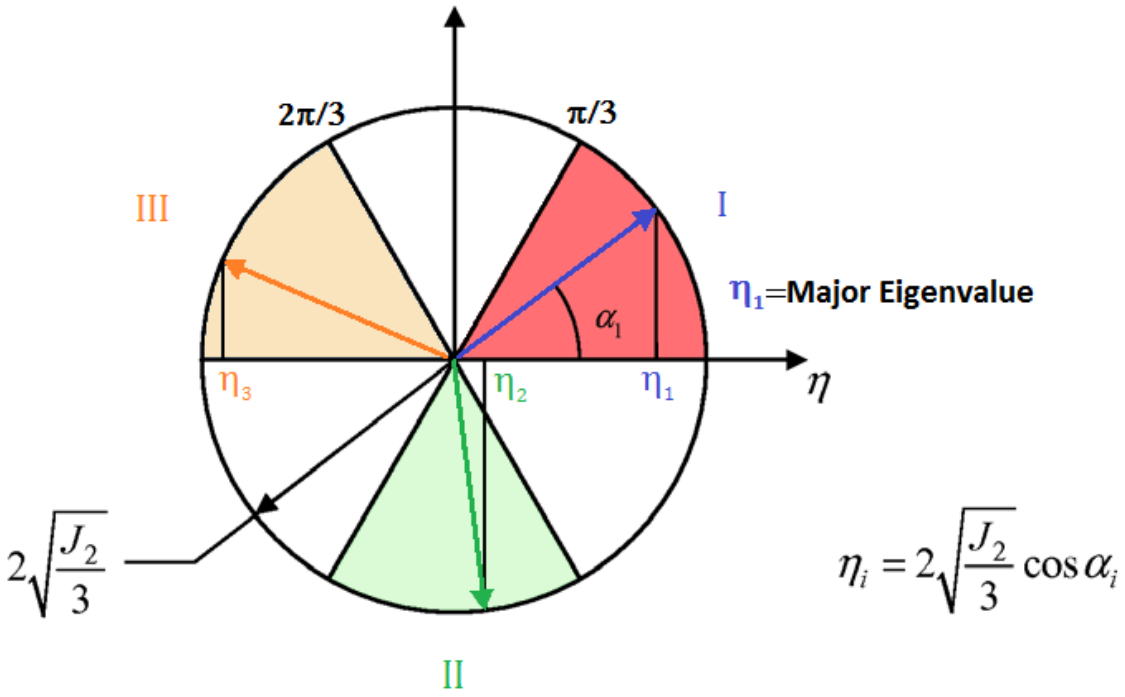


Figure 34: Major eigenvector calculation

Step 3: Calculate angle α_1: $\cos(3\alpha) = j_3/2\ ((3/j_2)^{3/2})$ Due to the nature of the data, (noncircular series of connected points), the major η will occur in the first quadrant. $0' <= \alpha_1 <= \pi/6$.

Step 4: Solve the major η value: $\eta_1 = 2\ \mathrm{sqrt}(j_2/3)\ \cos(\alpha_1)$

Step 5: Calculate the eigenvalue: $\lambda_1 = \eta_1 + 1/3\ \mathrm{tr}\mathbf{A}$

Now that we have the major eigenvalue, we can calculate the eigenvector (Equation 3) through simple substitution: **A v** = λ **v**, **A**= 3x3 covariance matrix, λ =known scalar value, solve for the 3x1 eigenvector **v**

$$\mathbf{A\ v} = \lambda\ \mathbf{v} = \begin{bmatrix} \mathrm{cov}_{xx} & \mathrm{cov}_{xy} & \mathrm{cov}_{xz} \\ \mathrm{cov}_{yx} & \mathrm{cov}_{yy} & \mathrm{cov}_{yz} \\ \mathrm{cov}_{zx} & \mathrm{cov}_{zy} & \mathrm{cov}_{zz} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Equation 3: Eigenvector calculation A=covariance matrix, v=eigenvector, □=eigenvalue

Step 1: $\lambda\ v_1 = a_{11}\ v_1 + a_{12}\ v_2 + a_{13}\ v_3$, solve for v_1, v_1=f(y,z)

Step 2: $\lambda\ v_2 = a_{21}\ v_1 + a_{22}\ v_2 + a_{23}\ v_3$, substitute the v_1 of step1, v_2=f(z)

Step 3: Set v_3=1, solve for v_2 in step2.

Step 4: Solve for v_1 in step1.

Step 5: $\lambda v_3 = a_{31} v_1 + a_{32} v_2 + a_{33} v_3$, solve for v_3. This extra step reduces the round-off errors. We are solving for the Principal Direction of the fiber segment which is the normalized (unit magnitude) version of the eigenvector. The normalized Principal Direction vector is saved for the next processing step.

A single bundle of 1293 covariance matrices will be converted to Principal Direction vectors in approximately 3.1mSec. The equivalent serial approach takes 265mSec.

5.3 Principal Direction Vector To Feature Score

This final process reduces the bundle of data to a single number. This number is used to compare the bundle against the reference library. The result is a score of how well the subject bundle matches the reference bundle.

```
GpuPrincipalDirToFeatureScore <<<1,
3*numPolarsRoundedUpToWarpsize,..>>> ();

// one bundle per grid, one block, each thread works on a polar coordinate (x, y, z)
GpuPrincipalDirToFeatureScore() {
    for(segmentIdx=0;  segmentIdx<segmentsInBundle;  segmentIdx++) {
        Check principal direction = general direction of the segment
        Calculate difference: abs( principal direction - polar )
        if (magnitude(difference) < polarRegionRadius)
            polars feature count+=1;
    }
    Convert feature count table to feature density table
    Normalize the feature density table
    Calculate library difference: abs(library feature - normalized feature)
    Calculate feature score: sum library difference for every polar
    if (feature score < current minimum score)
        Current minimum score = feature score;
}
```

Code block 7: Pseudo code of the GpuPrincipalDirToFeatureScore process

This process (Code block 7) converts the unit-vector Principal Direction (tracePoint) data to a set of feature or polar location counts. This count-set is then compared against the reference bundle's count-set. The difference between the two sets is the feature score.

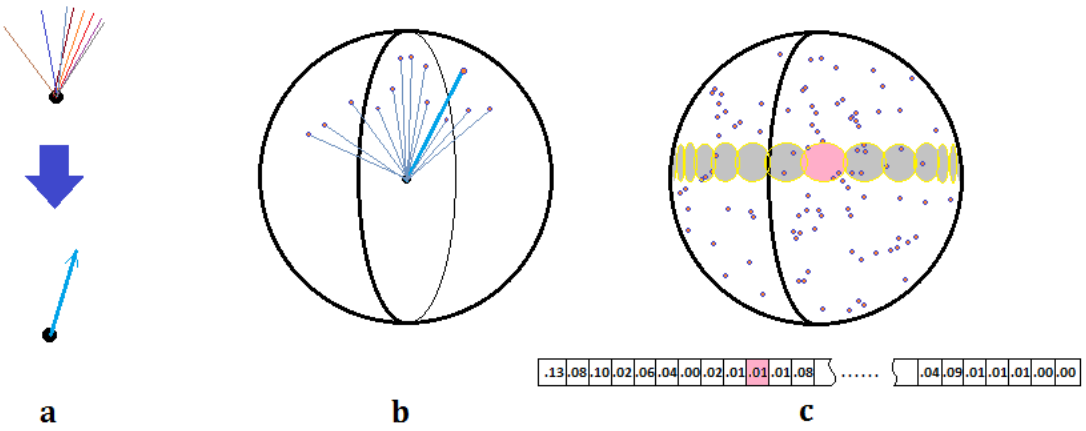


Figure 35: Feature Score (a) Segment of fiber points reduced to the Principal Direction, trace point (b) Single fiber of trace points (c) Bundle of trace points with a ring (only one shown) of Polar regions, and the resulting featureCount table

A sphere is divided into 62 polar regions. Each polar represents a location on the unit-sphere. The set of Principal Direction vectors (tracePoints) are compared against the Polars. If the unit-vector is inside the Polars polar-region, the feature-count for that polar increases. The feature-count is the number of Principal Direction vectors that point in this common direction. It is very much like a histogram for common directions. A single tracePoint may exist in more than one polar region. There are a fixed number of Polars, and a variable number of tracePoints.

Each pass through this process compares a single tracePoint against all 62 polar regions. The resulting feature count set is converted to a tracePoint density set (Equation 4). The density set is then normalized to give a percentage distribution set (Equation 5). The resulting table shows

how many fiber segments in the bundle are oriented in one direction versus any other direction. Bundles with similar fiber paths will have similar feature sets.

$$d_i = \frac{featureCnt_i}{nTracePts}$$

Equation 4: Feature Density calculation

$$f_j = \frac{d_j}{\sum_{i=1}^{nPolars} d_i}$$

Equation 5: Feature Density Normalization calculation

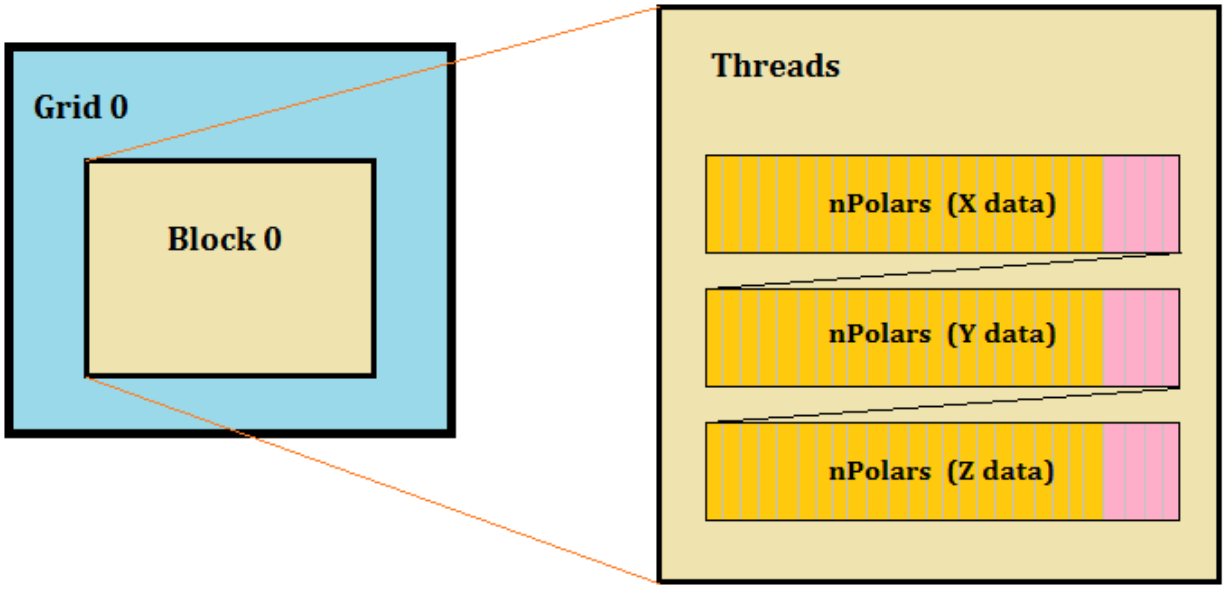


Figure 36: Principal Direction to Feature Score Grid Breakdown (Orange= valid data, Pink= unused threads)

This process uses a single block grid. (See Figure 36) (The Principal Direction data is in a single data-set.) Each thread in the block represents a single (XYZ) coordinate for each polar point on the unit-sphere. Each pass through the process compares a single Principal Direction unit-vector against every polar point. At the end of the process the number of active threads is reduced to the number of Polars to process the feature count data.

The first part of this process uses 3 threads (one for each coordinate) to verify the Principal Direction is in the same general direction as the fiber. If needed, the Principal Direction vector will be rotated 180'. This is done by applying a dot-product multiplication on the Principal Direction vector and the fibers general direction vector (calculated earlier). If they are out of phase, the Principal Direction vector is rotated.

The single Principal Direction vector is then compared against all polar regions (Equation 6). If the vector falls within the polar region, that Polar's feature count is increased. The polar regions are circular regions centered on the polar location on the unit sphere. (See Figure 35c) Because these polar regions overlap one another, a single Principal Direction vector may add to more than one feature count.

$$featureCnt \mathrel{+}= (abs(\overrightarrow{principalDir} - \overrightarrow{polarLoc}) < polarRegionRadius) \; ? \; 1 : 0$$

Equation 6: Polar Feature Count calculation

The resulting feature table is then converted to a feature density table. (Reference Equation 4) The number of active threads is reduced to the number of Polars. Each feature count is divided by the number of Principal Direction vectors in the fiber bundle. The density table is then normalized to produce a percentage polar distribution table. (Reference Equation 5) This distribution table is compared against the ROI's feature table. The difference between the table entries is summed for the final featureScore (Equation 7).

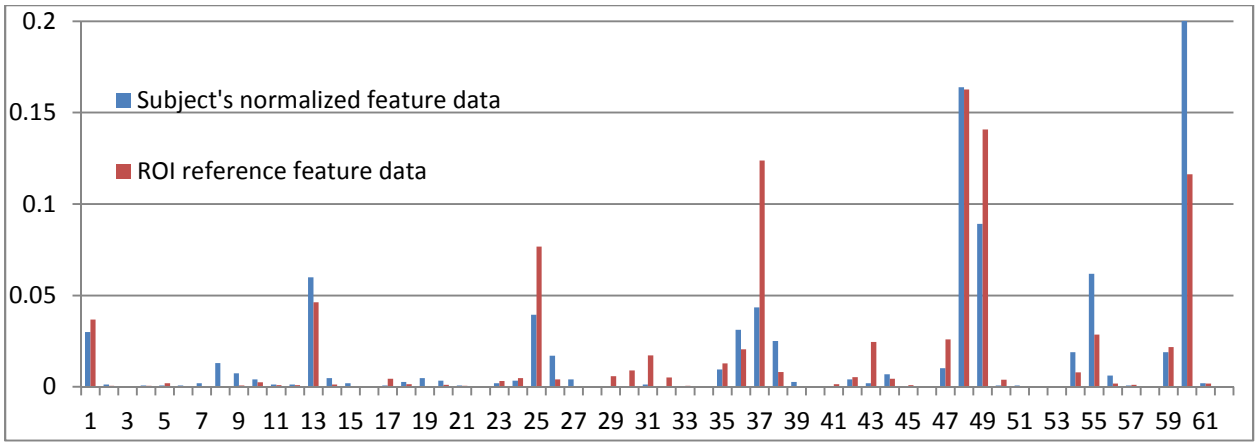


Figure 37: Feature score comparison of each Polar

$$featureScore = \sum_{i=0}^{nPolars} abs\left(feature_{norm_i} - feature_{ref_i}\right)$$

Equation 7: Feature Score calculation

A single bundle of 1293 Principal Direction vectors is compared against 62 Polars, then scored against the preprocessed ROI's feature-score, in approximately 2.6mSec. A 214KByte bundle of fiber data is reduced to one single-precision number in approximately 5.9mSec.

5.4 Bundle Concurrency

CudaDoTraceMap processes a single bundle of fibers. The internal processes (GpuBundleToCovariance, etc.) operate consecutively in the CudaDoTraceMap process. A single bundle is processed with each pass of the CudaDoTraceMap loop. Each bundle is processed in a separate stream, allowing each bundle to process asynchronously to the others. To reduce the launch time, the bundle processes are launched from a queue of statically allocated resources. Each stream uses its own GPU resources and executes independently of each other. The number of streams (GPU resources) used has been adjusted to allow all bundles in the ROI to run concurrently. This keeps the CPU and GPU as busy as possible. (Between different ROI's, the CPU spin-waits.)

5.5 Reference Feature Data Correction

DICCCOL works by comparing a subject's data against a reference library. The feature results of the reference library are preprocessed and are imported for comparison at the end of CudaDoTraceMap for calculation of the match score. This comparison works as long as the fiber data is processed in the exact same approach as the preprocessed reference library.

The approach I used is slightly different from the original code which uses 144 Polars. Many of these Polars are duplicates. My approach eliminates the duplicates and uses only the 62 unique Polars. To correct for the difference in Polar-count, the preprocessed feature data is renormalized. Because of this difference in approach, the numerical results of the comparison stage are different between the two approaches.

$$f_{corr_j} = \frac{f_{ref_j}}{\sum_{i=0}^{nPolars} f_{ref_j}}$$

Equation 8: Reference Library, feature correction calculation

Initially, all ROI feature results are loaded to the GPU at the start of the predictROI2 process. The data is then renormalized by a single grid (reference Equation 8). Only these unique preprocessed feature results are saved for the feature-count comparison. This kernel has not been optimized for speed, as it is a setup routine and it runs asynchronously to the CPU, during the setup stage. The execution time is typically less than the remaining CPU setup time. (Typically completes before the CPU has issued its next GPU task.) Every block in the grid works on a single ROI data set. Each block uses 32 threads (warpSize). The heart of the renormalization is simple summation reduction operation (reference Code block 8).

```
sumReductionTechnique() {
    if(threadIdx < warpSize) {
        for(idx=threadIdx+warpSize;  idx<lastData;  idx+=warpSize)
```

```
            warpSum+= sharedMemory[idx];
        sharedMemory[threadIdx]= warpSum;
    }
    __syncthreads();    // wait for single warp reduction to complete
    for( idx=warpSize>>1;  idx>0;  idx>>=1) {
        if(threadIdx<idx)
            sharedMemory[threadIdx]+=  sharedMemory[threadIdx+idx];
        __syncthreads();    // wait for sharedMemory to finish write cycle
    }
    If(threadIdx == 0)
        totalSum= sharedMemory[0];
}
```

Code block 8: Pseudo code of the summation reduction technique

The CPU->GPU transfer of the complete feature data set (100KByte) takes approximately 24uSec. The complete data set is renormalized in 85uSec.

At the beginning of every ROI loop, that ROI's feature results are moved from global memory to constant memory for faster access. This GPU internal 288Byte move takes a negligible amount of time.

5.6 Polar Point Generation

The last phase of the CudaDoTraceMap process count's the number of Principal Direction vectors that fall within predefined polar regions. In order for the comparison to be valid, the Polars must be located in the exact positions and order as the preprocessed feature results used. These matching Polars are generated by the GPU, and stored in constant memory for fast access.

This kernel runs asynchronously and has not been optimized for speed. This task occurs once during the predictROI2 setup stage. The Polars are generated by a single block grid. Each thread in this block generates a single polar. (See Figure 38)

Generating the 62 unique polar locations take approximately 2.5mSec.

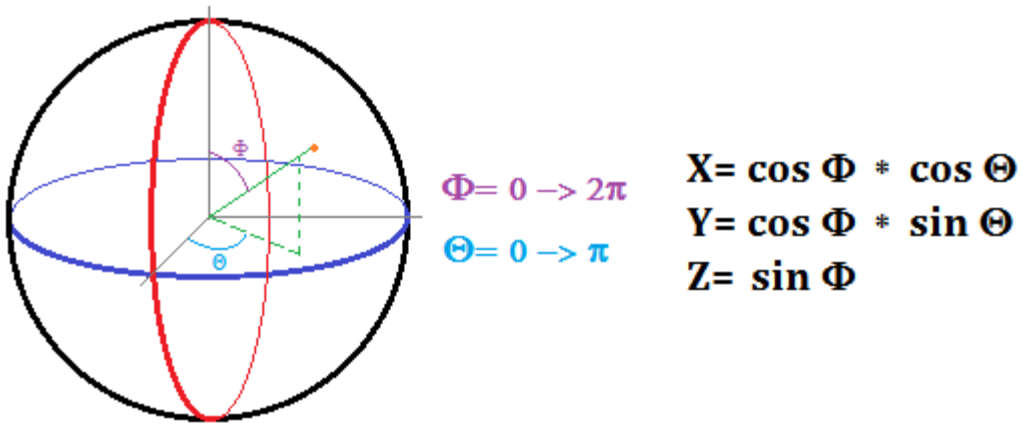


Figure 38: Polar Locations calculation

6. Testing

The eigenvalue and eigenvector calculations were tested for accuracy and precision. A side-by-side visual comparison of each subjects ROI 0 results was performed. The list of resulting best-match bundle ID's was compared between the serial and parallel approaches. A comparison of the best-match bundles was performed to identify identical fiber content. The number of generated segments from the CPU process was compared to the GPU process. A visual comparison between the two approaches was performed on a subset of ROIs for a single subject. A serial versus parallel run-time comparison was performed to determine the resulting speedup. The machine details can be found in section 11. Machine/System Details.

6.1 Eigenproblem Comparison

The calculation of the eigenvalue and eigenvector is a new approach. Under certain conditions[4], the eigenvalue precision breaks down. Due to the nature of the predictROI2 task, these conditions don't occur here. To prove the approach and verify the precision, the results of my calculations were compared against the CULA[3] Library (a parallel version of the Linear Algebra Package library) results. The sparse symmetric eigenproblem solver, culaDeviceSsyev(), was used to generate the eigenvalues and eigenvectors from the covariance matrix of subject 9. These single-precision results were compared against my calculated results. The single Principal Direction vector results were subtracted from the corresponding CULA results (reference Table 3). The test shows the numbers to be identical to 4 digits of precision. DICCCOL only needs a single digit of precision.

6.2 Serial vs Parallel Quantitative Comparison

The serial versus parallel DICCCOL results were compared by examining the bundle ID's of the individual ROI's. I compared the GPU bundle ID results against the CPU bundle ID results.

The number of ROIs that yield the same bundle result as well as different bundle result is counted. Table 4 shows the comparison of the best-match bundle for the two approaches.

I compared the bundle contents for a subset of ROIs in subject 9. The subject was chosen at random. The comparison was limited to a subset of the ROIs because it is a time intensive process. Table 5 shows the number of fibers in the CPU results that are also in the GPU results. The differences come from how the fibers were filtered. The original serial approach ignored fibers that were less than a half-segment in length, but only fibers that were at least a one segment in length, 16 fiber-points, would actually generate any data. My parallel approach ignores fibers that are less than two segments, 24 fiber-points, in length. (See discussion in section 9. Conclusion)

I compared the segment counts from the fibers of a single bundle. Table 6 shows the fiber count comparison for bundle 574 in ROI 0 of subject 9. The subject, ROI, and bundle selection were chosen at random. The fiber count discrepancies are caused by the differences in the segmentation process. The original serial approach missed a segment in some of the fibers.

Table 6 shows the segment count of the individual fibers in the first bundle of ROI 0. It demonstrates one of the process differences between the CPU approach and the GPU approach. When there is a difference, the GPU process extracts one more segment than the CPU process. See Code block 9 for a comparison between the two segment generation approaches. The original approach uses "<" for the loop comparison which drops the last segment when the fiber size divides evenly by the segment size. This is corrected when "<=" is used as the loop comparator.

```
SEG_INTERVAL = 8
SEG_SIZE= SEG_INTERVAL << 1
```

New approach: (CTraceMap::CudaDoTraceMap())

```
segCnt= ((nFiberPts<<1) - SEG_SIZE) / SEG_SIZE
```

Original Approach: (CTraceMap::TracingSingleFiber())

```
int currentFiberSize= currentFiber.size()
for(int segStart=0, segEnd=SEG_SIZE;
                segEnd<currentFiberSize;
                        segStart+=SEG_INTERVAL, segEnd+=SEG_INTERVAL)
```

New Approach	
nFiberPts	segCnt
31	2
32	3
33	3

Original Approach	
currentFiberSize	seg
31	2
32	2
33	3

Code block 9: Comparing Segment Counts

6.3 Single ROI Visual Comparison

A side-by-side visual comparison of the CPU vs GPU ROI 0 results was performed. The images with obvious differences were noted. ROI 0 was selected at random for this comparison. (See Figure 40)

6.4 Multiple ROI Visual Comparison

Figure 41 compares a subset of 48 ROI's from a single subject. The CPU generated images were compared against the GPU generated images for subject 12. The ROIs with major differences were noted. The subject 12 was chosen at random for this comparison.

6.5 Runtime Comparison

Timing results are provided by a CPU timestamp comparison. A timestamp is taken at the beginning and end of the predictROI2 routine. (See Code block 10) The difference, in seconds, is reported as the execution time. (See Code block 11) A minimum of 3 runs were performed on both the CPU & GPU code. The average of these runtimes (see Table 2) is used for the calculated Speedup factor. See Table 8 for the timing results.

predict.pipeline (script file)

```
echo "** cleanup last **"
# cleanup (last run) task runs here

echo "** reistration **"
before= "$(date +%s)"
# registration step runs here
regTime="$(expr $(date +%s) - $before)"

predict.pipeline  (script file)
echo "** prediction **"
before= "$(date +%s)"
predictROI2 arg1,arg2,arg3,..
predTime="$(expr $(date +%s) - $before)"

echo "** prepare for viewer **"
before= "$(date +%s)"
# viewer step runs here
viewTime="$(expr $(date +%s) - $before)"

echo
echo "registration time:  $regTime sec"
echo "prediction time:  $predTime sec"
echo "viewer time:  $viewTime sec"
```

Code block 10: DICCCOL Command Pipeline (with timers)

Sample run: (cpu, subject 9)

```
predict.pipeline
arg1,arg2,arg3,…
** cleanup last **
** registration **
** prediction **
** prepare for viewer **

registration time: 17 sec
prediction time:  656 sec
viewer time:       0 sec
```

Code block 11: Sample CPU Run of DICCCOL, Subject 9

Table 2: Average Runtime Calculation example, Subject 9

Run	1	2	3	4	5	6	7	8	9	10	**Average** (seconds)	Std Dev	Speedup
CPU	652	652	656	662	667	653	648	648	672	660	657.0	7.67	6.42
GPU	102	103	102								102.3	0.47	

7. Results

7.1 Eigen Calculation Accuracy

The single precision calculation of eigenvectors shows 4 digits of precision. A single precision number is limited to 7 decimal digits. Table 3 shows the absolute error for the eigenvector and eigenvalue numbers when compared to the CULA generated numbers. nSegments is the number of fiber segments that were processed. The eigen calculation is performed once per segment.

Table 3: Absolute error with the CULA generated results and the number of eigen calculations performed

Subject	9	6
eVector X	0.000016034	0.000076115
eVector Y	0.000008741	0.000010395
eVector Z	0.000000632	0.000000477
eValue	0.000019073	0.000020981
nSegments	20,955,341	13,836,770

7.2 CPU vs GPU Bundle Results

Table 4 compares the serial vs parallel ROI results. The CPU bundle ID for each ROI is compared against the GPU process results. For Subject 9: of the 358 ROI's there were 104 identical results and 254 different results. When the bundle ID changes this doesn't mean the resulting image drastically changes. See Figure 40 for comparisons.

Table 4: Comparing the CPU vs GPU bundle results

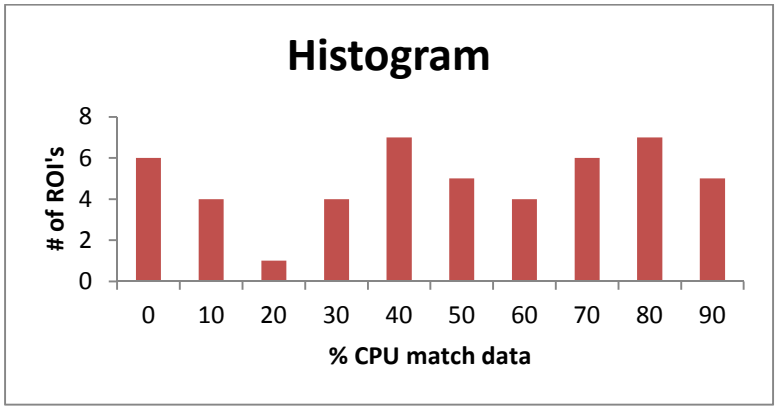
ROI Bundle Results: CPU vs GPU			
Subject	Same Bundle ID (count)	Different Bundle ID (count)	Change in Bundle ID results
1	117	241	67.3%
6	115	243	67.9%
9	104	254	70.9%
10	109	249	69.6%
12	108	250	69.8%
14	85	273	76.3%
16	85	273	76.3%
20	100	258	72.1%

7.3 CPU vs GPU Fiber Results

While the bundle ID's may be different for the CPU vs GPU results, the bundles may contain common fibers. Table 5 demonstrates this. It compares the CPU fiber results against the GPU fiber results. It shows how many of the fibers from the CPU bundle results are also in the GPU results. (The percent of CPU fibers that are common to both CPU and GPU result bundles.) A 100% result may mean the CPU and GPU bundle ID's match, or that all the fibers in the CPU result are also in the GPU result. It does not address fibers in the GPU result that are not in the CPU result. Figure 39 shows a result where there are no (0%) CPU fibers in the GPU result.

Table 5: Subject 9 fiber bundle comparison (first 100 ROI's): ROI # vs % of CPU fibers that are also in the GPU result bundle

ROI #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
% match	26.3	100.0	33.3	100.0	100.0	26.9	87.2	100.0	60.2	100.0	62.5	100.0	88.4	100.0	100.0	100.0	55.0	76.0	0.0	25.4
	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	95.7	31.0	4.9	100.0	32.7	77.5	100.0	8.3	100.0	100.0	100.0	100.0	100.0	34.0	94.1	46.2	100.0	61.5	100.0	76.9
	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
	100.0	100.0	73.1	80.9	62.5	98.2	80.6	38.8	100.0	51.0	0.0	100.0	100.0	100.0	100.0	63.6	100.0	34.8	100.0	100.0
	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	0.0	100.0	100.0	47.1	47.8	100.0	100.0	100.0	0.0	1.3	0.0	13.9	100.0	100.0	100.0	100.0	100.0	0.0	100.0	100.0
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99
	100.0	46.3	53.8	93.8	53.7	74.5	34.8	75.6	82.6	30.0	71.4	100.0	95.0	48.2	66.7	100.0	100.0	93.5	8.5	100.0



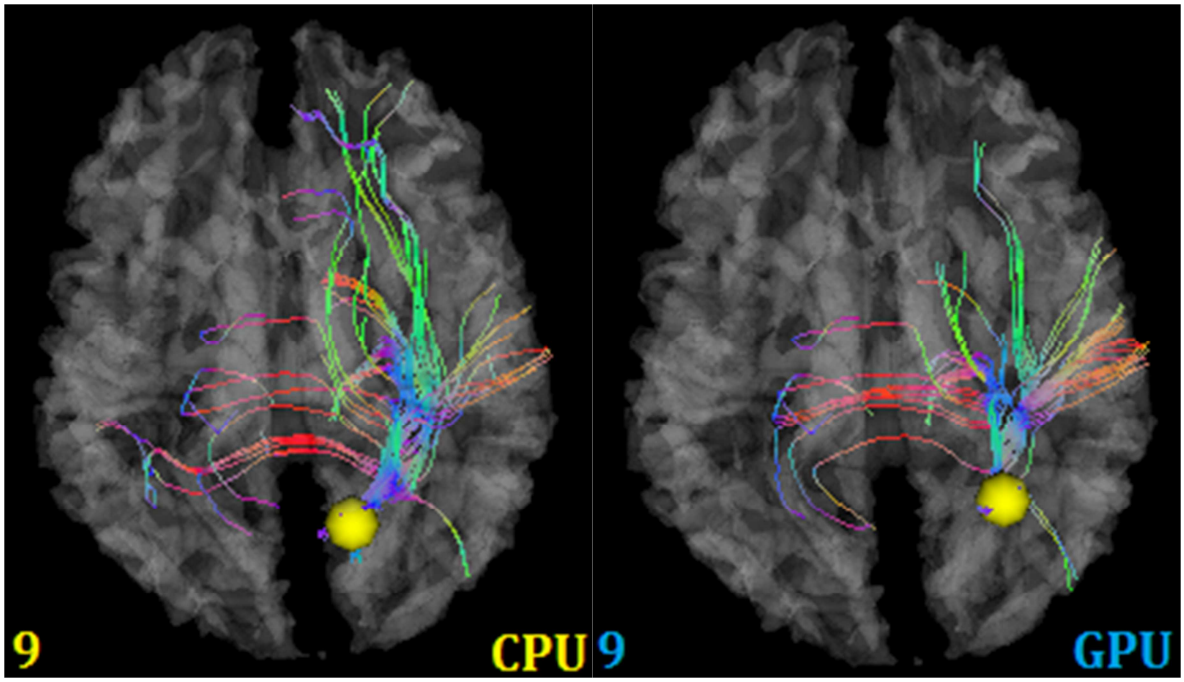


Figure 39: Subject 9, ROI 60: Completely different results
(CPU: bundle 9391, GPU: bundle 11938)

7.4 Fiber Segment Count Results

Table 6 shows the segment count of the individual fibers in the first bundle of ROI 0. Of the 95 fibers in this bundle, 13 generated a different number of segments.

Table 6: Segment Count of Fibers in Bundle 574, ROI 0, Subject 9

Fiber Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU	14	13	12	13	12	12	8	11	11	11	20	13	8	12	8	9	9
GPU	14	13	12	13	12	12	9	12	11	11	20	13	8	12	8	9	10
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
	9	10	11	11	8	7	12	12	8	9	11	12	4	3	11	12	3
	9	11	11	11	8	8	12	12	8	9	11	12	4	3	11	12	4
	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
	9	3	13	10	8	10	10	19	11	9	9	11	2	11	12	17	14
	9	3	13	10	8	10	11	19	11	9	10	11	2	11	12	17	14
	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
	18	19	3	10	20	18	4	9	23	18	12	12	10	19	19	19	18
	18	19	3	10	20	18	4	9	23	19	12	12	10	19	19	19	18
	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84
	18	21	16	18	20	19	18	17	19	17	19	18	18	18	19	19	19
	18	21	16	18	20	20	19	17	19	17	19	18	18	18	19	20	19

85	86	87	88	89	90	91	92	93	94
17	17	18	18	17	17	23	24	17	21
17	17	18	18	18	17	23	24	17	21

7.5 CPU vs GPU ROI-0 Visual Comparison Results

Of the 13 subjects used for testing, 7 of the CPU results visually matched the GPU results. The remaining 6 comparisons showed very similar results.

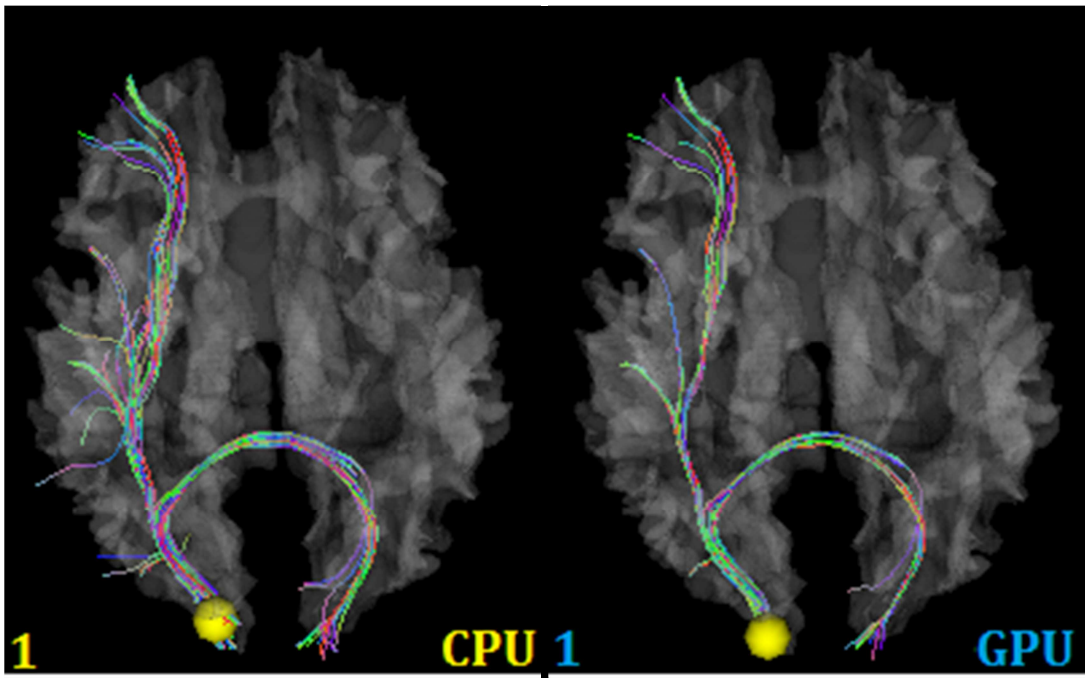


Figure 40-1: Subject 1, ROI 0: Results differ[i] (CPU: bundle 491, GPU: bundle 149)

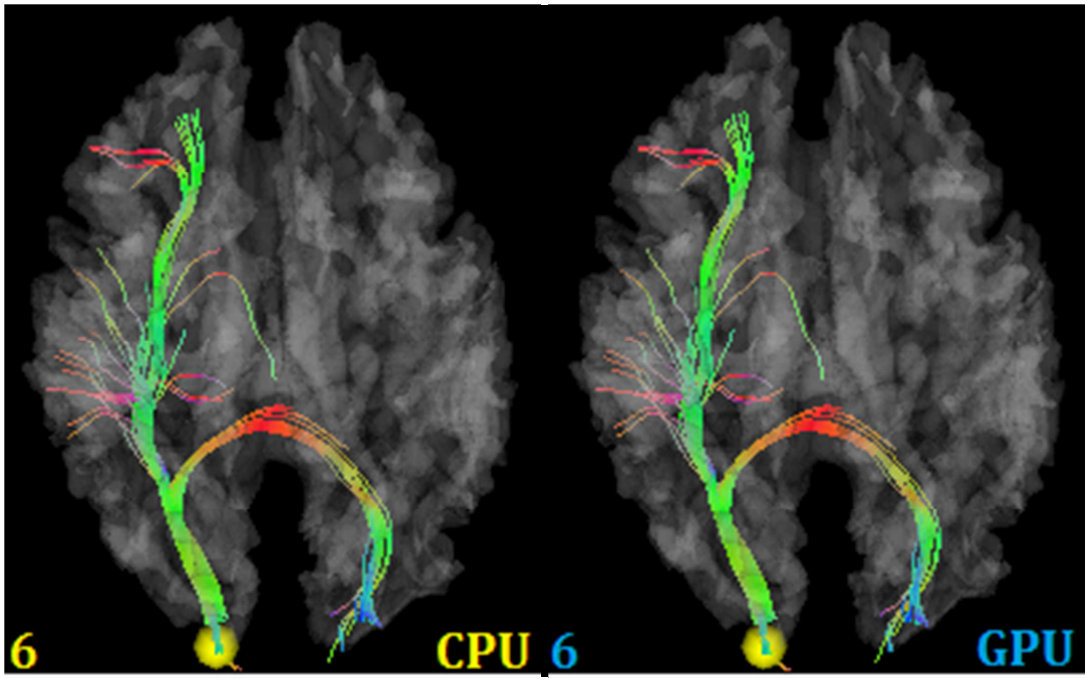


Figure 40-2: Subject 6, ROI 0: Results match (CPU: bundle 75, GPU: bundle 75)

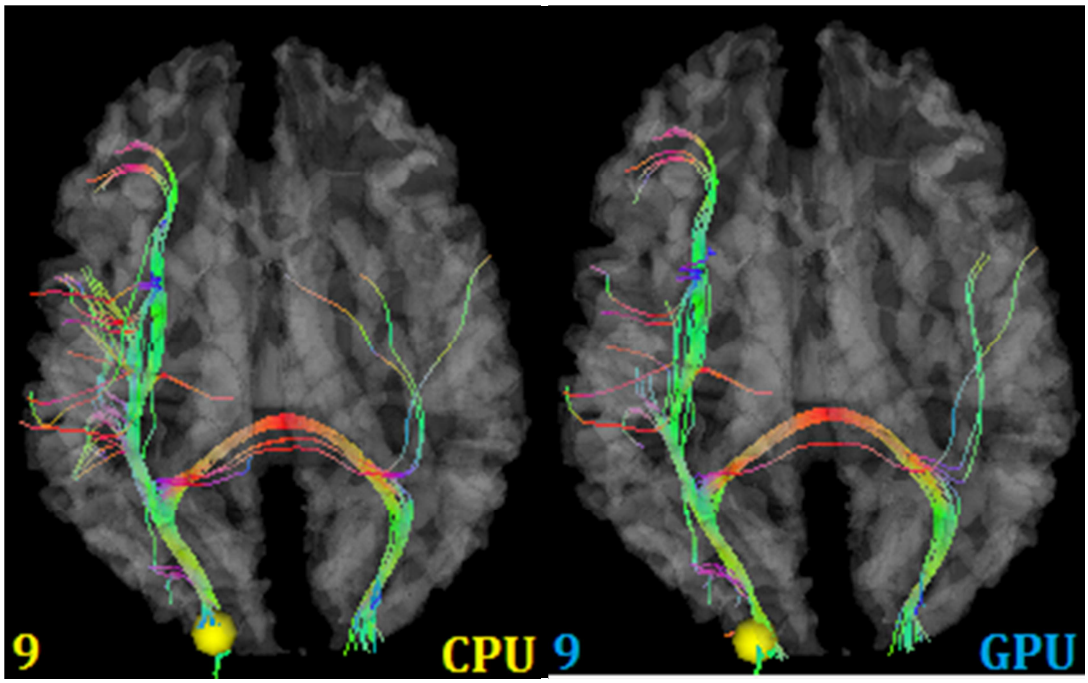


Figure 40-3: Subject 9, ROI 0: Results differ (CPU: bundle 574, GPU: bundle 304)

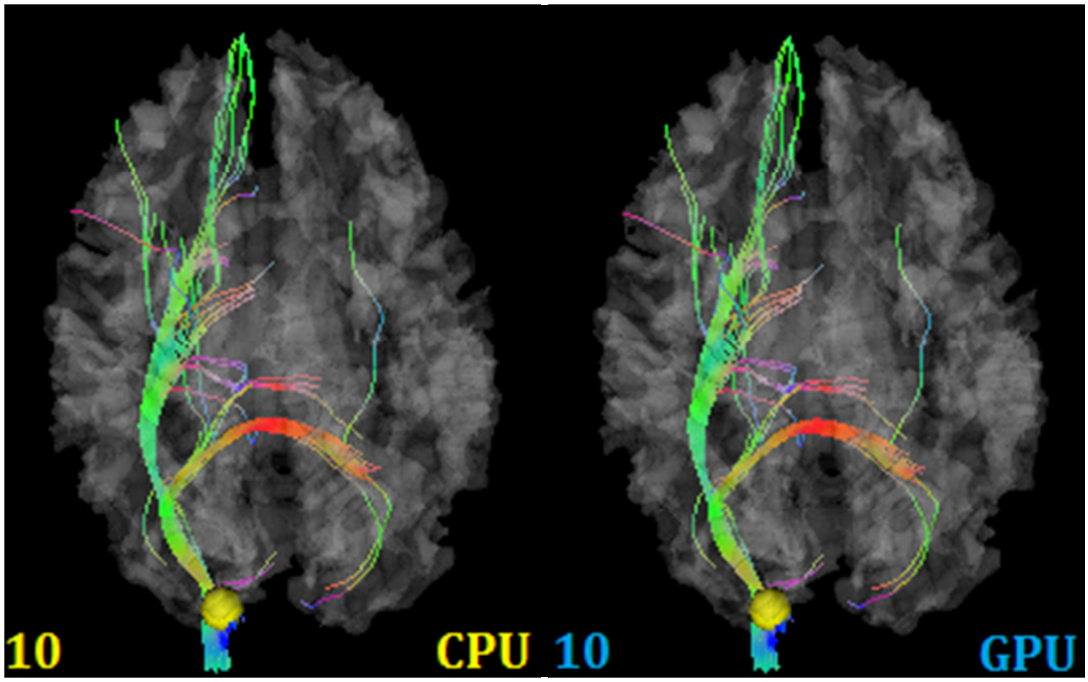


Figure 40-4: Subject 10, ROI 0: Results match (CPU: bundle 871, GPU: bundle 871)

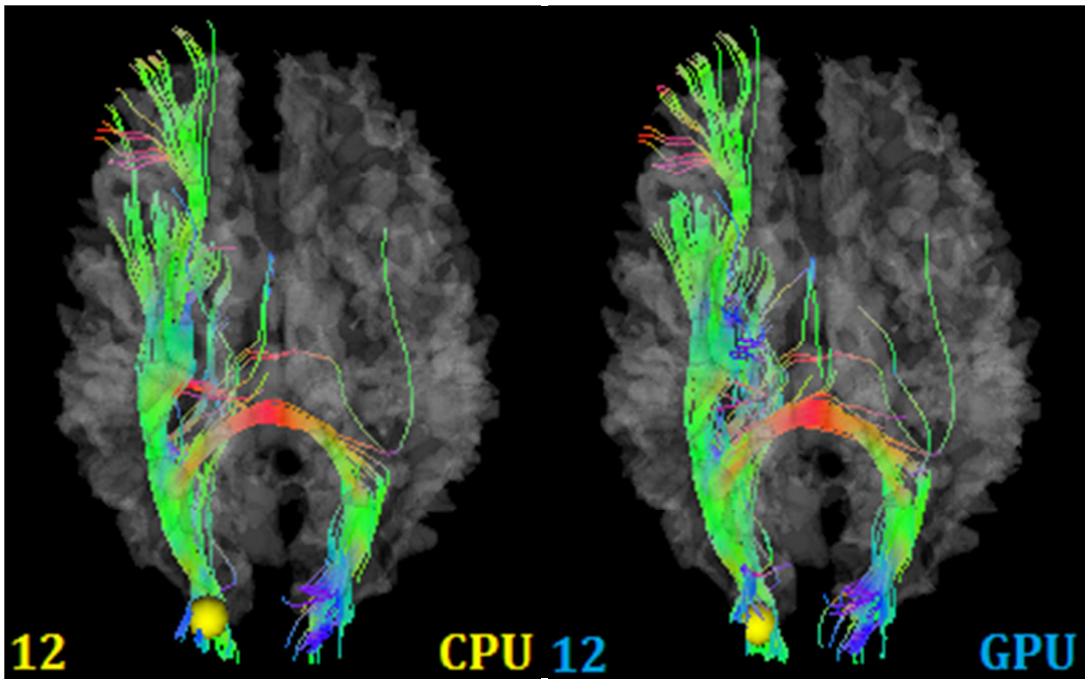


Figure 40-5: Subject 12, ROI 0: Results differ (CPU: bundle 232, GPU: bundle 106)

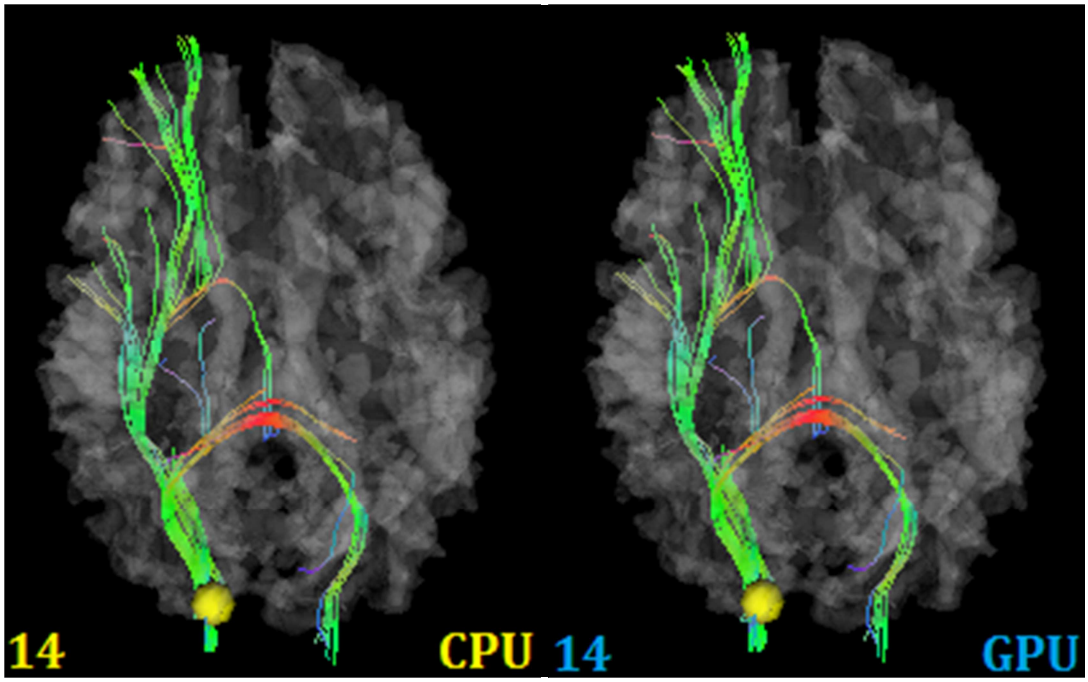


Figure 40-6: Subject 14, ROI 0: Results match (CPU: bundle 891, GPU: bundle 891)

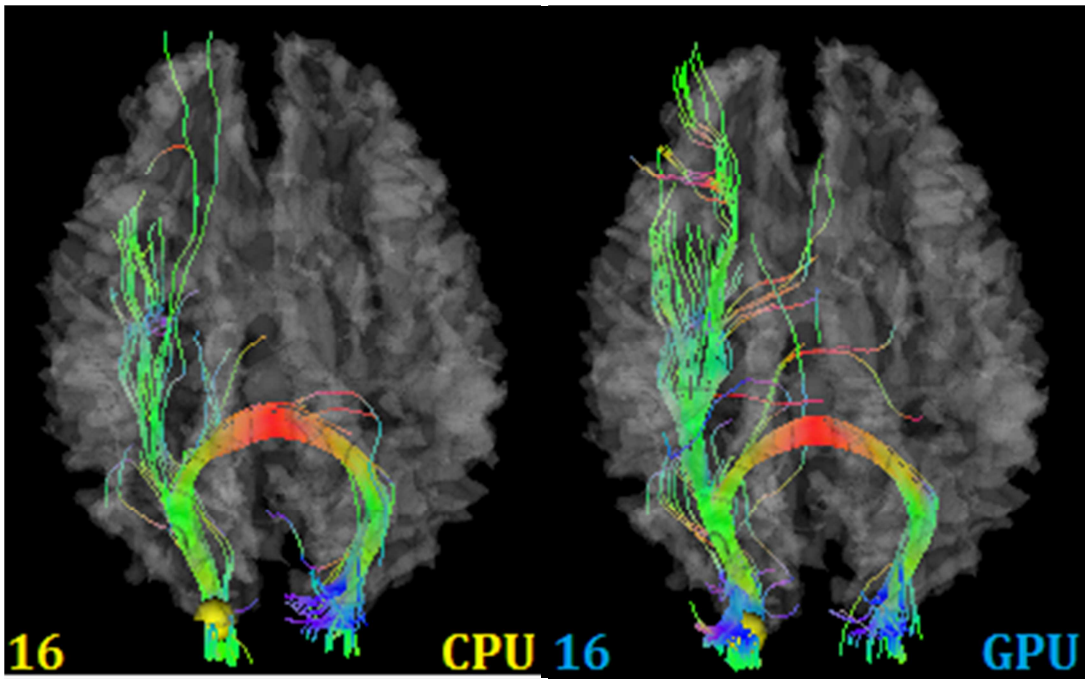


Figure 40-7: Subject 16, ROI 0: Results differ (CPU: bundle 77, GPU: bundle 5)

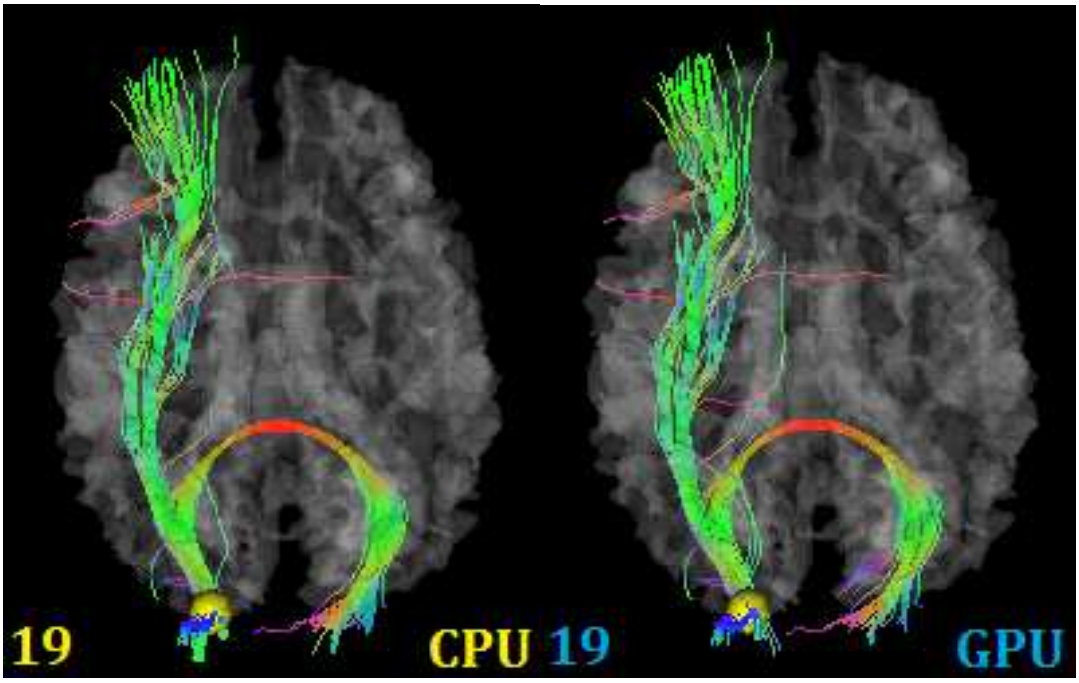


Figure 40-8: Subject 19, ROI 0: Results differ (CPU: bundle 318, GPU: bundle 326)

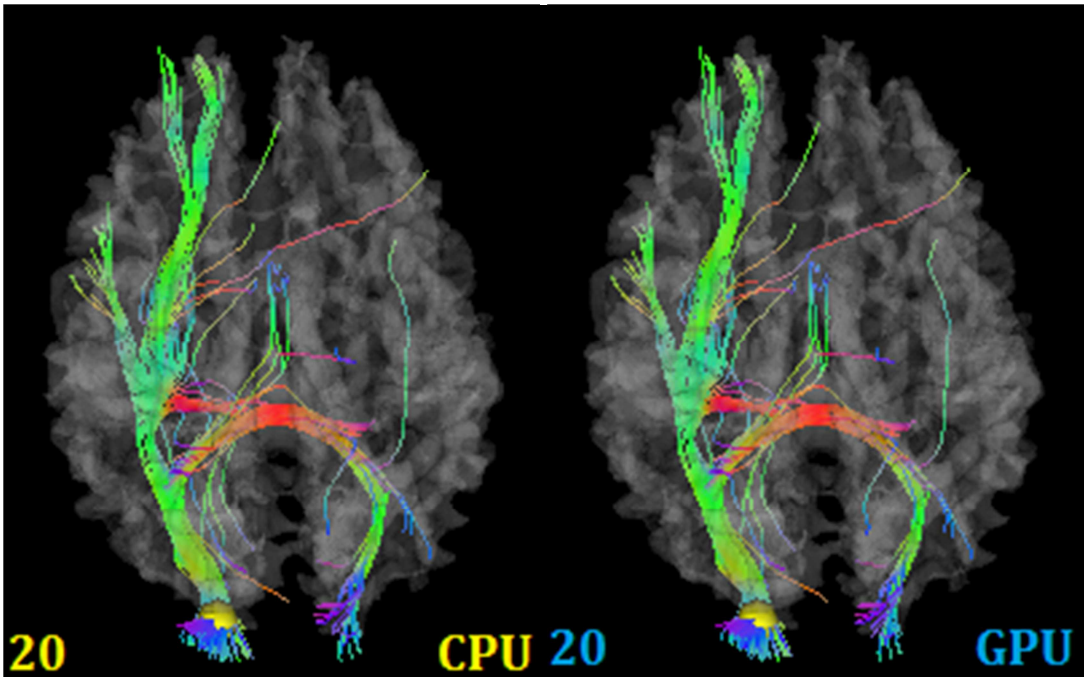


Figure 40-9: Subject 20, ROI 0: Results match (CPU: bundle 62, GPU: bundle 62)

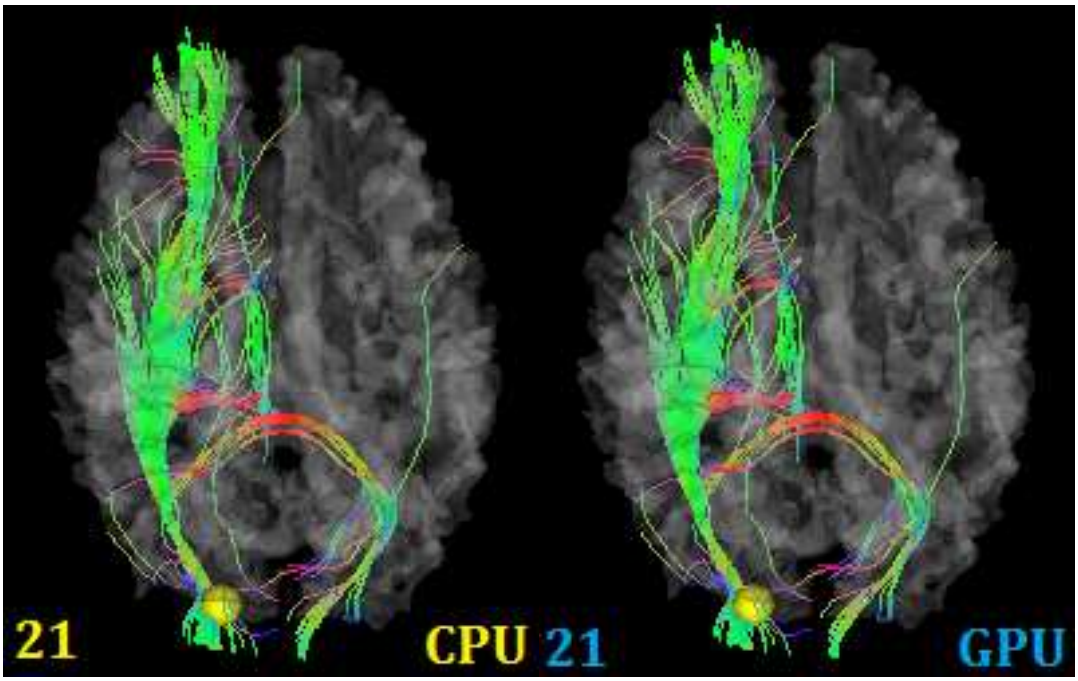


Figure 40-10: Subject 21, ROI 0: Results match (CPU: bundle 674, GPU: bundle 674)

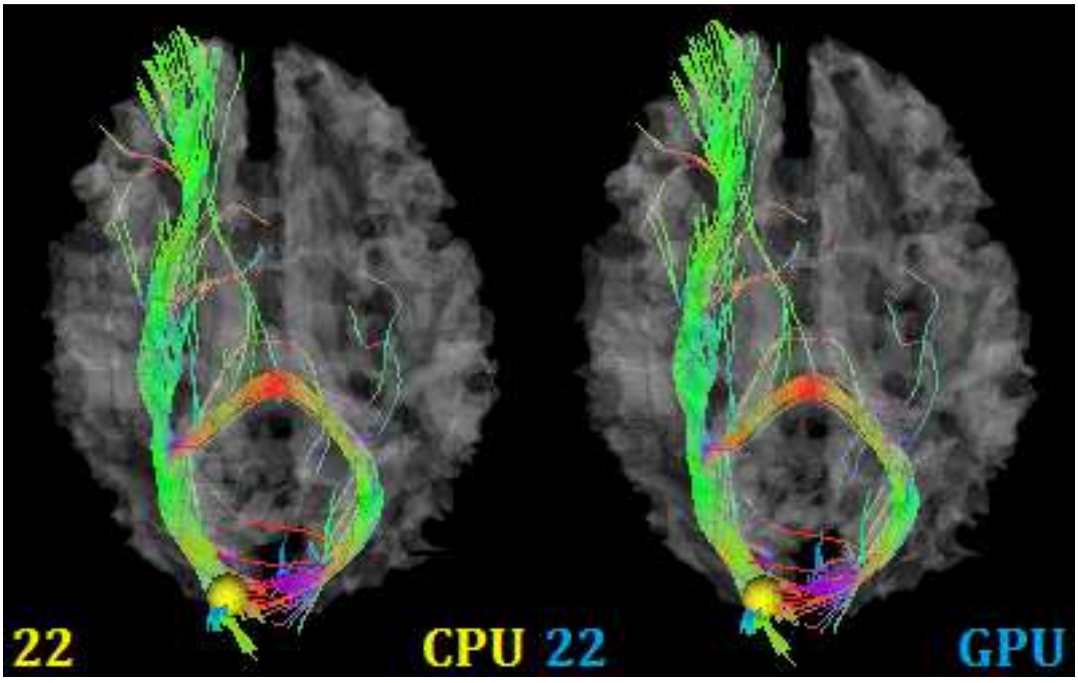


Figure 40-11: Subject 22, ROI 0: Results match (CPU: bundle 191, GPU: bundle 191)

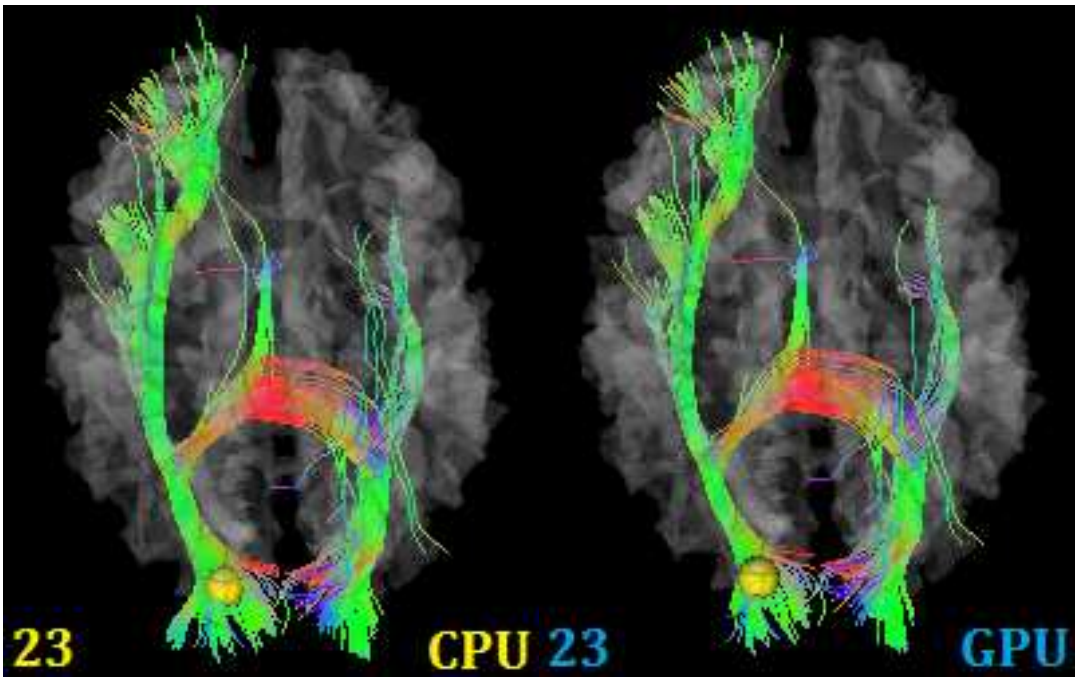


Figure 40-12: Subject 23, ROI 0: Results differ (CPU: bundle 404, GPU: bundle 654)

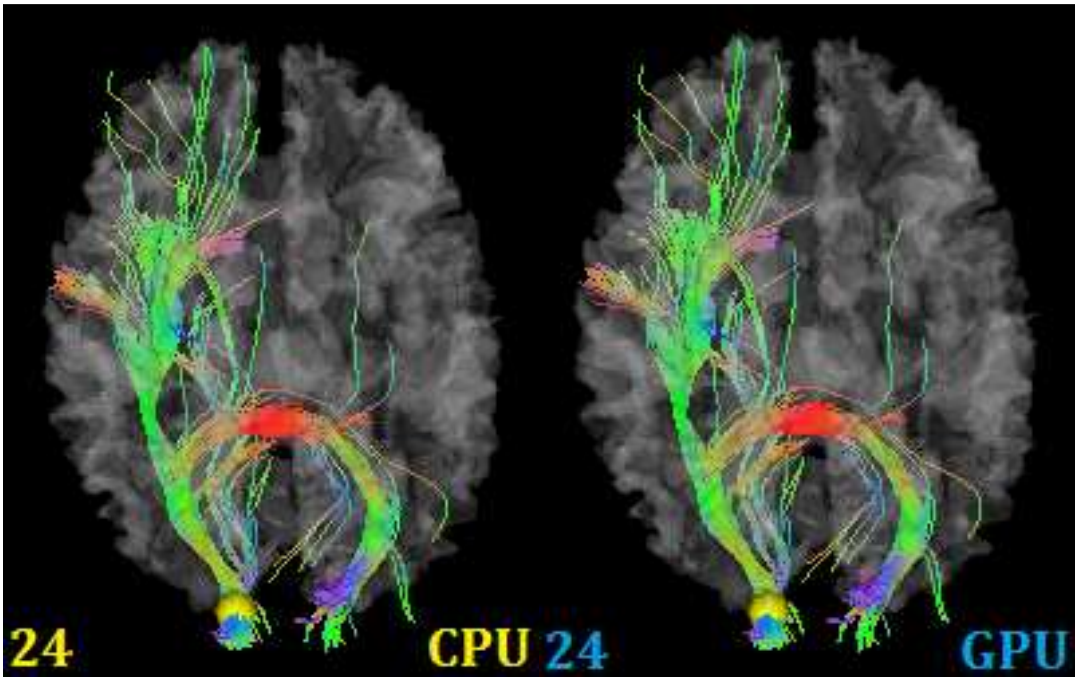


Figure 40-13: Subject 24, ROI 0: Results match (CPU: bundle 132, GPU: bundle 132)

7.6 CPU vs GPU Single Subject Visual Comparison Results

The CPU generated images are displayed with a yellow frame (upper), and the corresponding GPU images have a blue frame (lower). It is suggested to use the zoom function in the electronic version of this document for better viewing. The ROI's are displayed in sequence with ROI 0 at the top left and ROI 48 at the bottom right. Most of the ROI comparison's show no or small differences. The ROI's that show the most dramatic differences are: 3, 14, 39, 41. Less dramatic differences can be found with ROI: 2, 18, 24, 47.

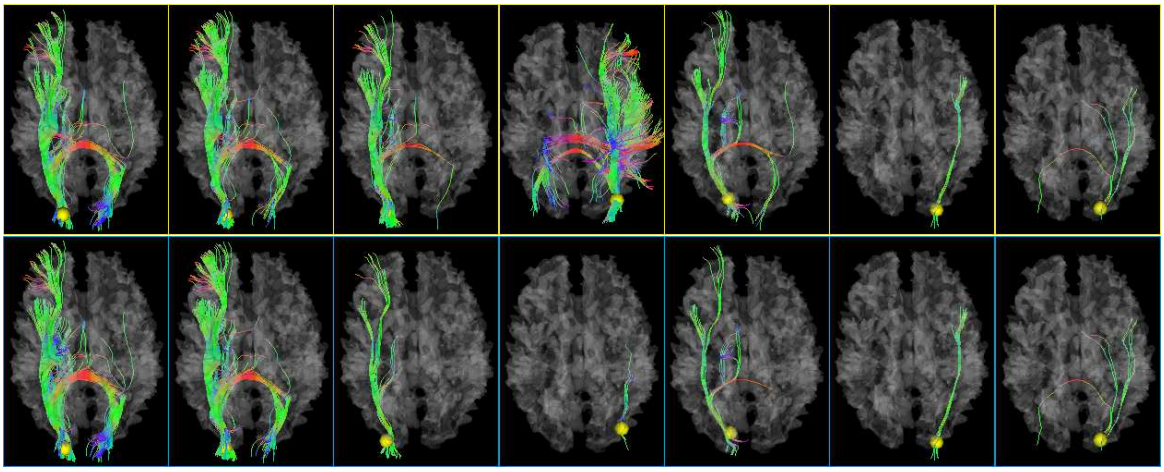


Figure 41: Subject 12, ROI 0(left) – 6(right): CPU Results(yellow, upper) GPU Results(blue, lower)

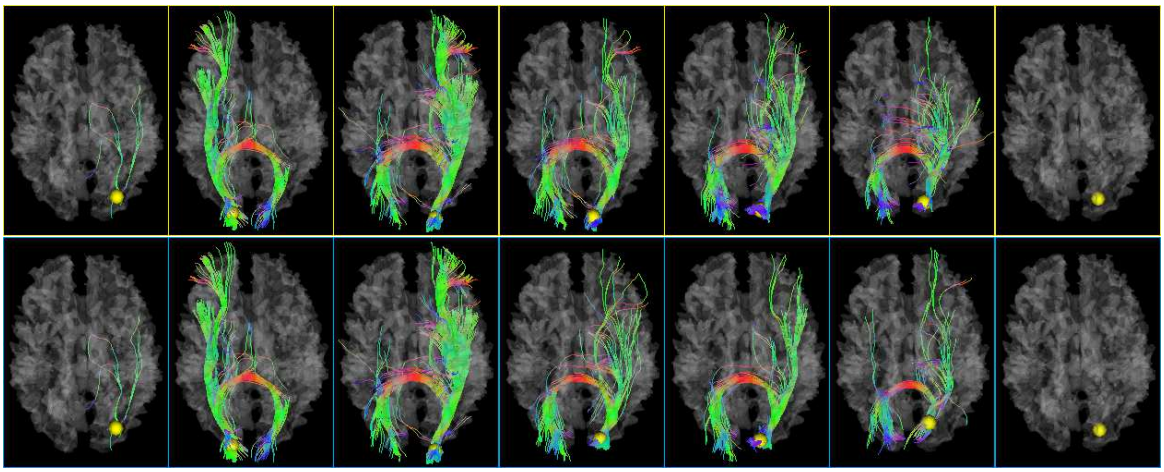


Figure 41: Subject 12, ROI 7(left) – 13(right): CPU Results(yellow, upper) GPU Results(blue, lower)

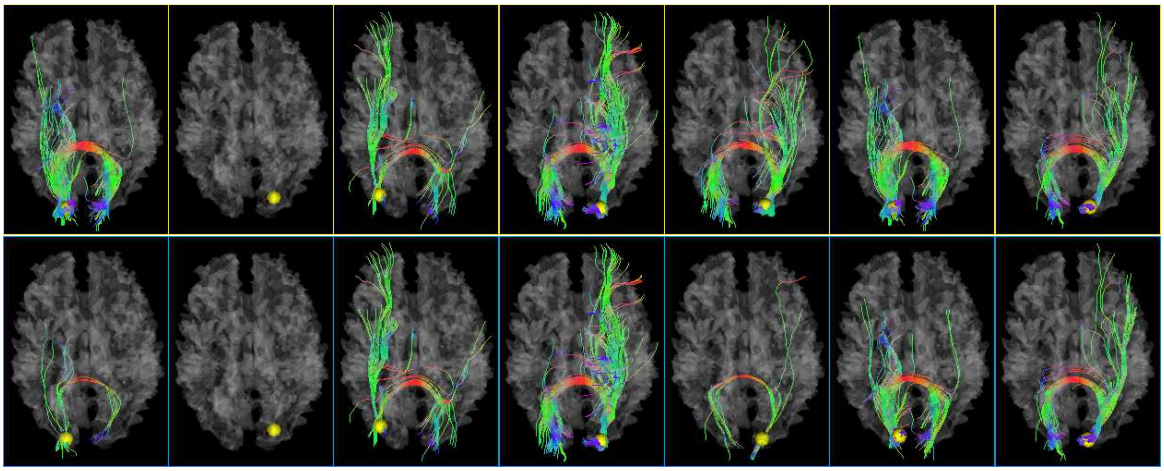


Figure 41: Subject 12, ROI 14(left) – 20(right): CPU Results(yellow, upper) GPU Results(blue, lower)

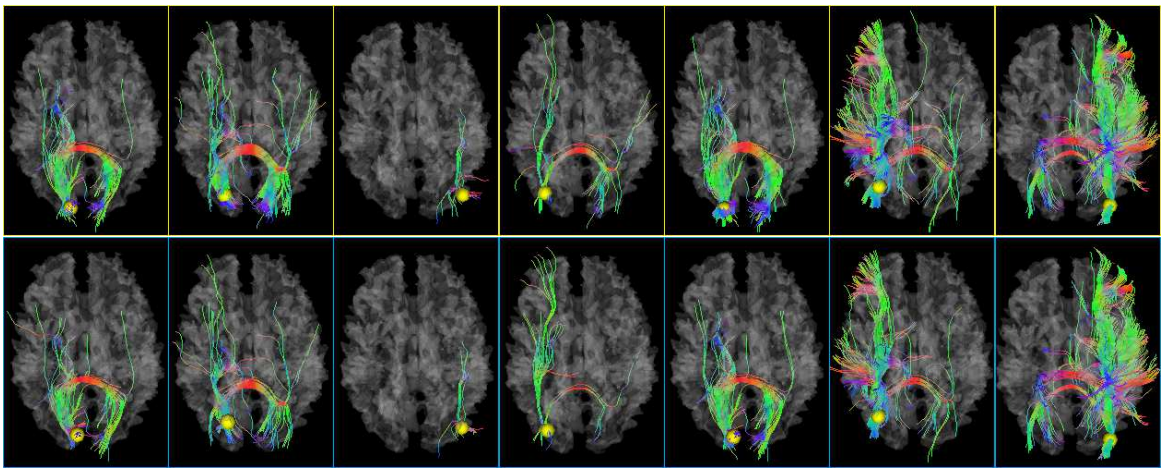


Figure 41: Subject 12, ROI 21(left) – 27(right): CPU Results(yellow, upper) GPU Results(blue, lower)

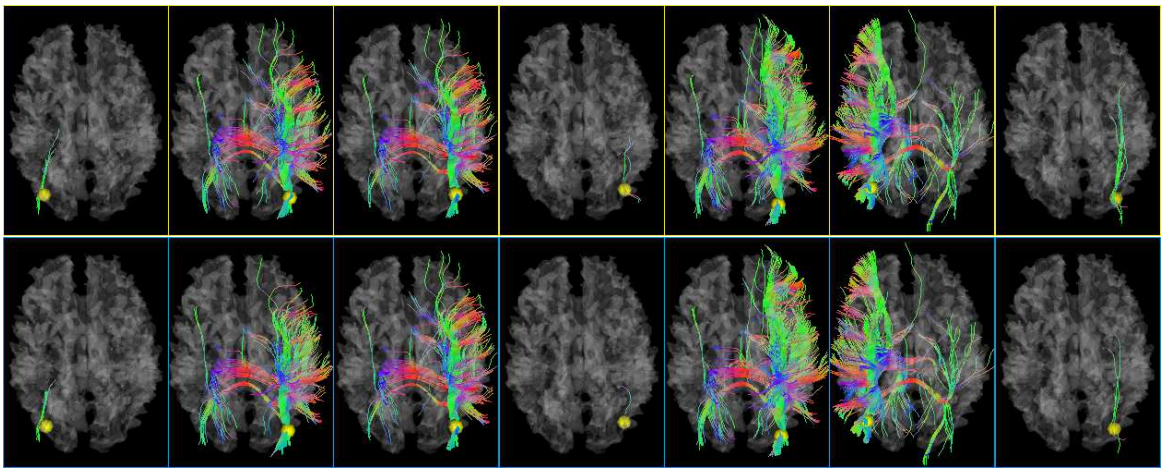


Figure 41: Subject 12, ROI 28(left) – 34(right): CPU Results(yellow, upper) GPU Results(blue, lower)

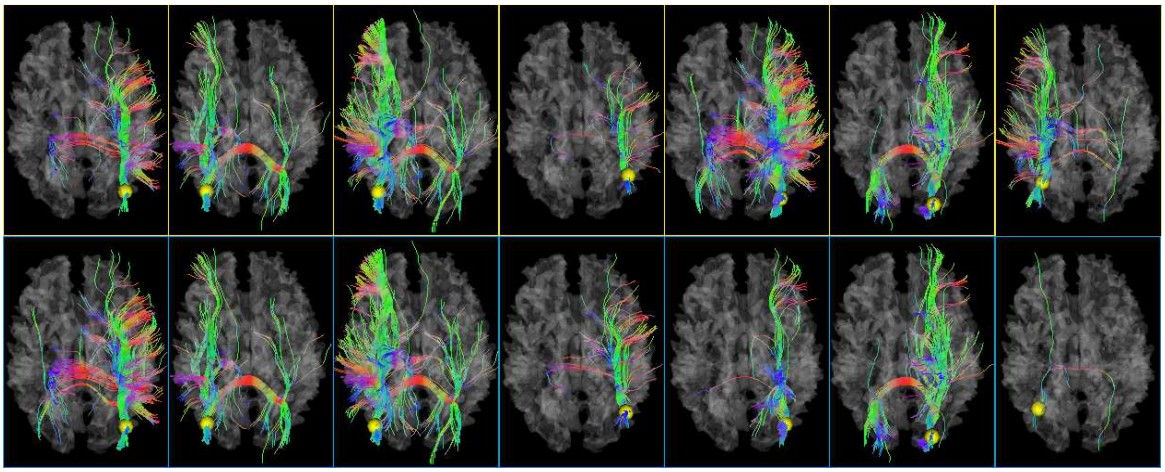


Figure 41: Subject 12, ROI 35(left) – 41(right): CPU Results(yellow, upper) GPU Results(blue, lower)

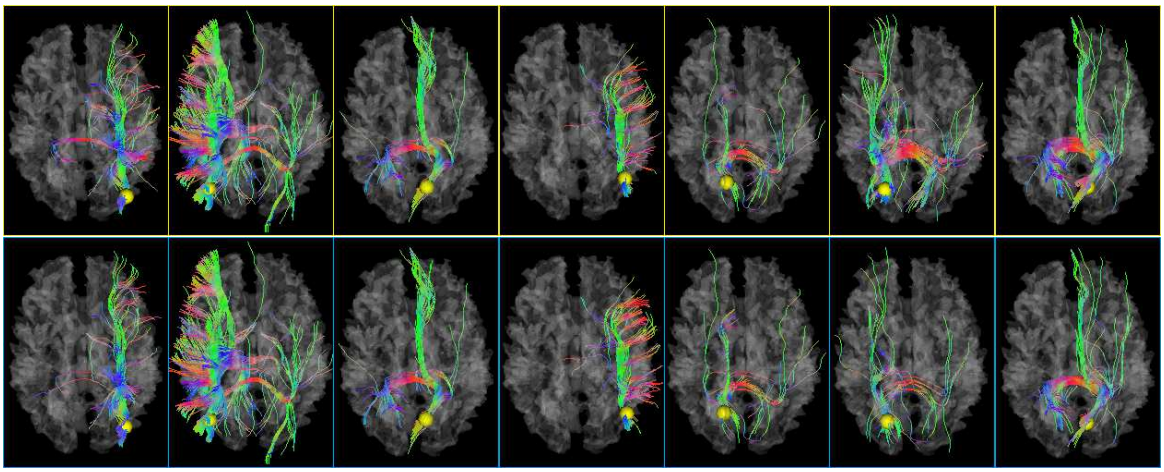


Figure 41: Subject 12, ROI 42(left) – 48(right): CPU Results(yellow, upper) GPU Results(blue, lower)

7.7 Process Runtime Result

Table 7 compares the serial vs parallel runtime of the three major DICCCOL processes. The Registration column is the FSL, FLIRT process. The Prediction column is the predictROI2 process. The Viewer column is the data collection process for the generateProfile4Viewer11 tool. (See Code block 10 and Code block 11)

Table 7: DICCCOL Process Runtime Comparison

		Single Runtime (seconds)		
	Subject	Registration	Prediction	Viewer
CPU	1	21	1664	0
GPU		20	250	0
CPU	6	22	437	0
GPU		22	72	0
CPU	9	17	655	0
GPU		17	103	0
CPU	10	14	1324	1
GPU		14	204	0
CPU	12	17	1564	0
GPU		18	241	0
CPU	14	17	1316	0
GPU		17	203	0
CPU	16	18	1977	0

GPU		18	303	0
CPU	19	15	1615	0
GPU		16	250	0
CPU	20	16	1411	0
GPU		17	218	0
CPU	21	17	1859	0
GPU		17	288	0
CPU	22	17	1579	0
GPU		17	245	0
CPU	23	18	1445	0
GPU		17	228	0
CPU	24	15	1813	0
GPU		16	282	0

7.8 PredictROI2 Runtime Results

Table 4 compares the serial vs parallel runtime of the PredictROI2 process. A minimum of three tests were run on each subject to provide the average runtime results.

Table 8: DICCCOL Runtime Comparison

	Average Runtime (Sec)				
	CPU		GPU		Speedup
Subject	Std Dev			Std Dev	factor
1	7.12	1687.6	254.0	0.00	6.6
6	2.43	439.3	72.0	0.00	6.1
9	7.67	657.0	102.3	0.47	6.4
10	4.99	1332.7	204.3	0.47	6.5
12	4.78	1553.7	241.7	0.47	6.4
14	0.47	1300.7	203.0	0.00	6.4
16	9.68	1948.8	302.0	0.00	6.5
19	4.12	1616.2	251.0	0.71	6.4
20	2.16	1408.0	219.0	0.00	6.4
21	11.58	1879.0	288.3	0.43	6.5
22	10.52	1595.8	245.8	0.43	6.5
23	15.66	1476.2	228.3	0.43	6.5
24	18.74	1841.3	282.3	0.43	6.5

8. Observations

Much of the number processing for the predictROI2 routine involves generating average values. Calculating an average involves a summation and division process. The summation process is serial in nature and the division process is parallel in nature. Because of this, neither the CPU nor the GPU processor is a great platform for performing the task of averaging. The averaging operation can be performed by either processor, but neither processor is capable of ideally performing both the serial and parallel aspects of the averaging operation. As long as the data set is many times larger than warpSize of the machine, the GPU can perform the operation faster than the CPU can. A warpSize reduction technique is used in many places of this GPU predictROI2 code. The data set is reduced to a single warp, and then a repetitive half-warp routine reduces it to a single value. This proved to be a very useful tool for the summation process, but care must be taken to address the inherent data hazards of the code. Properly addressing the data hazards corrected the numerical results and further reduced the runtime.

The CPU is not slowing down the process. A faster CPU will not make this process run significantly faster. The CPU spins idle while the GPU processes the bundles in bursts of work. As a result, the GPU is not being fully utilized. The GPU does a good job at swapping threads efficiently, but the predictROI2 processes is rather simple. I found that a single-warp (warpSize) process ran faster than a block that uses multiples of warpSize. When the code was changed to allow for multiple warps, the execution time increased. With proper data-coalescing this simple task ran faster on a single warp, than it did with a multiple warp execution.

CULA[3] is a parallel coded version of the popular Linear Algebra Library, LAPACK. CULA solves high-dimension eigenproblems rather quickly. The CULA Library is not considered a viable solution for my speedup goal. The use of the tool breaks the concurrency approach used

here. Without a different process approach, the CULA approach doubles the execution time. It ran slower than the CPU approach. The CULA Library is a useful tool when solving hundreds of eigenproblems with 100's of dimensions. The 3-dimension Principal Direction task used here is much too simple for this tool to be useful for my runtime goals. I would need more than a million Principal Direction problems in a single bundle before I could expect to see a speed improvement with this tool.

The eigenproblem calculation was the most time consuming operation in the original predictROI2 process. It used double precision numbers, and calculated all 3 eigenvalues and eigenvectors. The problem doesn't require high precision or a solution of every eigenvector. The covariance matrix is a small (3 dimension), symmetrical matrix. Only a single precision result is needed. The problem requires the solution of only the Principal Direction (major) vector. My solution takes advantage of all these simplification and provides a vector that offers 6 digits of precision.

The final stage of the predictROI2 process compares processed Subject bundles against preprocessed library results. This essentially is a comparison of direction-sorted results. This comparison is only valid if the library data and the subject data are processed in identical fashion. This doesn't happen, for two reasons. Reason 1: In my approach, the Subject data has been optimized for minimal processing time. This includes ignoring fibers that are too short. Fibers that are less than 2 segments in length are dropped. I suggest a single segment fiber isn't long enough to generate useful information. Reason 2: An examination of the CPU approach shows that it misses certain fiber segments. (See Table 6) A single bundle comparison of Subject9, shows a difference of 13 tracePoints, (CPU: 1280, GPU: 1293). The data is different going into the feature count process. This suggests the results will be different

In the sort routine, the library data uses 144 Polars, of which only 62 are unique polar locations. My approach only uses the unique locations for the sort. An adjustment is made to the library data for the final comparison. It would be better if this adjustment wasn't necessary. Adjusting for this difference in Polars doesn't adversely affect the runtime of the process, but it does add a fair amount of code overhead. (This will be a maintenance issue moving forward.)

The CPU vs GPU results comparison (Table 4) shows the differences in bundle selection. The difference in bundle selection is expected to be minor as the visual results indicate. Identical results, though not expected, are possible.

9. Conclusion

In this paper I show how the DICCCOL task could be performed more efficiently by taking a parallel programming approach. My resulting algorithm reduces the average runtime by a factor of >6 for all test subjects. I am unaware of a quantitative test approach that will verify my results against the serial DICCCOL approach. My parallel DICCCOL results don't visually match those of the serial approach. The differences are the result of corrections to what I perceived as oversights in the serial approach.

My parallel approach extracts more data points from the provided DTI data than the serial approach does. Because of this added data I consider my results to be more accurate than that of the serial approach. The second difference of my parallel implementation involves the selection of the minimum fiber length. The minimum fiber length for the serial approach is one segment. The minimum fiber length for my approach is two segments. I suggest a single segment fiber is a malformed fiber in the subjects DTI data set and therefor it should be removed.

DICCCOL library is about correspondence between subject brains. DICCCOLs are landmarks with high correspondence across many subjects. This correspondence is identified by the connectivity of these landmarks. Connectivity to these landmarks don't appear to be local, but to other distant regions of the brain. Thus very short connections in white matter don't exist and are errors in the DTI data. This is related to the ability of the tractography field to quantify their results.

This is not a study of the serial DICCCOL results vs the parallel DICCCOL results. I am not able to prove my parallel approach is any more accurate than the original serial approach only that it executes much faster. I have shown the serial results can be similar but different from

those of the parallel results. The difference in results needs to be reviewed by others in the neurophysiology field.

If this approach is adopted the considerable time savings allows DICCCOL to be a more practical research tool for the field of neurophysiology. Other areas for improvement of the tool can be found in section 10. Future Work.

10. Future Work

Using the preprocessed library data is cumbersome. The subject and library data needs to be processed in identical fashion or the comparison is likely invalid. The library data (mdls.features) needs to be cleaned of the duplicate polar information. 62 Polars is a rather small number of buckets for this direction-sort approach. What the sort is effectively doing, is counting the number of fiber segments that point in a particular direction. Increasing the number of Polars increases the resolution of the bundle comparison. The existing code can accommodate up to 96 Polars with very little change and have a minor impact to the execution time. I had an early version of the 62 direction-sort process that ran in ~10mSec. When the number of directions was increased to 80,000, the execution time only doubled. When using more Polars the efficiency scales up dramatically, but the reverse isn't true. Using fewer Polars doesn't dramatically reduce the execution time. Adding more polar locations only helps if they are unique. The polar rings and the polar radius used here have varying degrees of overlap (aside from the duplication issue). There are also empty regions where tracePoints go uncounted. At the minimum, 7% of the total area is not covered by a polar region.

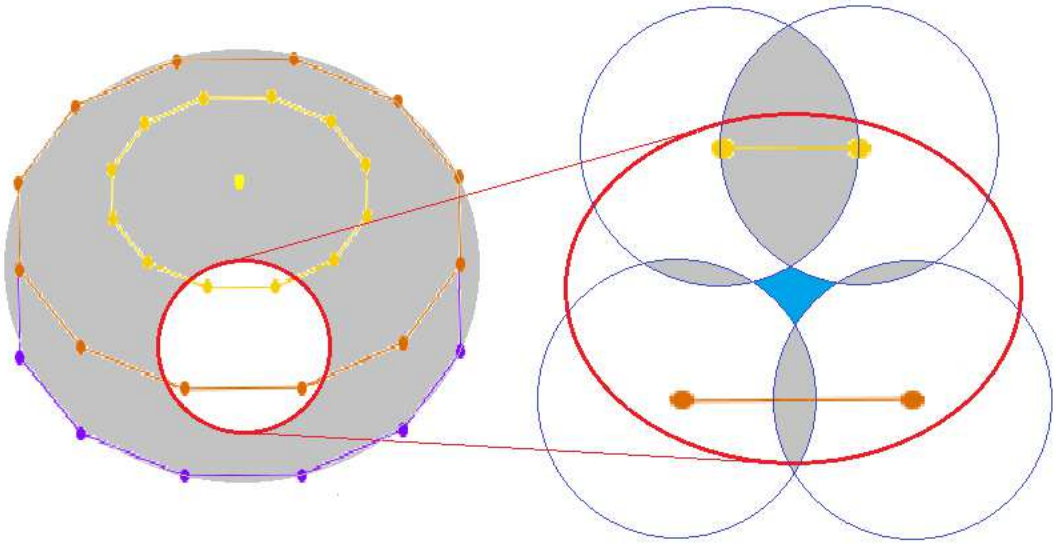


Figure 42: 62 Unique Ringed Polars Blue: Uncounted region Gray: Overlap (double count) region

This results in expensive segment data that is just ignored. There is no way of knowing if a significant cluster is located in this uncovered area. A better approach than this "ring of Polars", would be the use of equidistant[5] polar locations. With this approach the amount of polar region overlap is consistent and there is no overlooked data. The full data set is used in the final comparison. I tested this approach, but couldn't use it because the preprocessed library requires the use of the "ringed polar" approach.

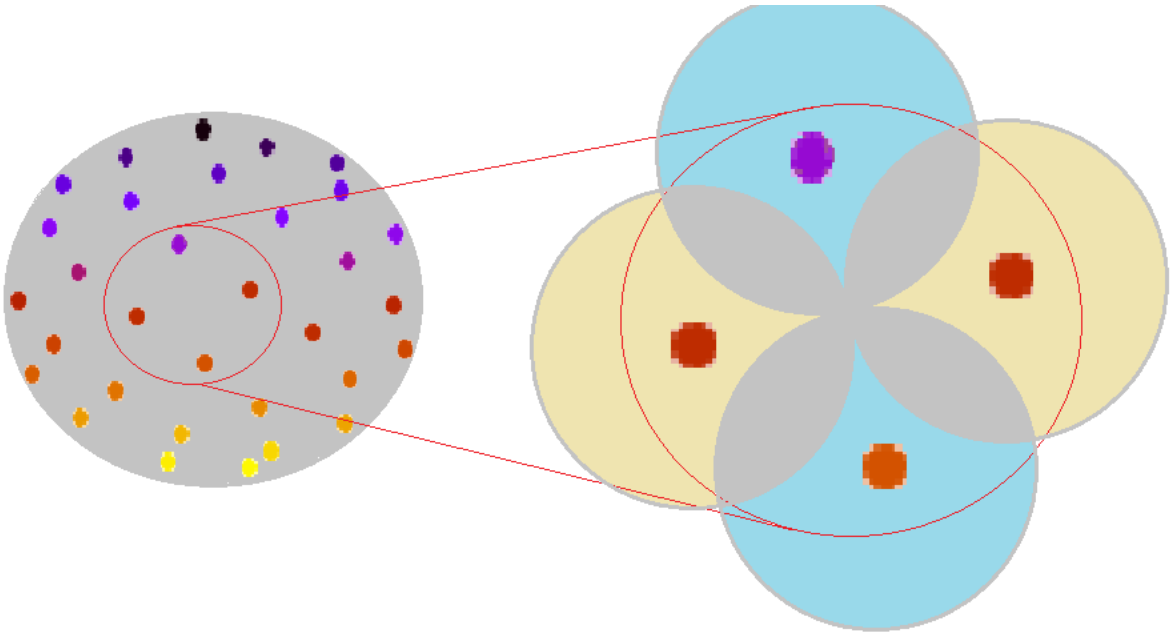


Figure 43: 62 Unique Equidistant Polars

The final comparison process performs a feature density calculation. Is the feature density calculation necessary? The next process normalizes the polar region feature distribution. The result is a polar-region percentage distribution table. For example, 5% of the fiber segments are oriented in the direction of polar J, while 42% go in the direction of the polar K. I suggest the density calculation can be removed as it only adds a constant multiplier to the Feature Density Normalization equation.

$$f_j = \frac{d_j \ \mathbf{nTracePts}}{\sum_{i=0}^{nPolars} featureCnt_i}$$

Equation 9: Feature Density Normalization calculation

The work done here addresses the low level operation of the predictROI2 task. The processing work is done in bursts of ROI subject bundles. To further reduce execution time, the separate ROI tasks should be issued concurrently. This requires rewriting the predictROI2, main2.cpp

file to allow concurrent ROI processing. Effectively, this would be a grid-computing approach on a single GPU.

To shorten the execution time of the existing code, one could employ a bigger GPU. The very small data set used by DICCCOL is transferred to the GPU very quickly. Data bus bandwidth is not slowing the process down. The GPU HW I used has 4 Streaming Multiprocessors (SM) (CoreConfig: 192:32:16). There are other more capable[6] GPU's that supports up to 14 SM's (CoreConfig: 2688:224:48). More SM's means the HW is capable of doing more concurrent work.

This code was developed on Nvidia's Fermi HW. Their next generation HW, Kepler, allows kernels to launch other kernels. This adds another level of flexibility for newer programming approaches.

The changes addressed by this paper can be applied to the serial process to significant improve the efficiency of DICCCOL. Correcting the logic of the fiber segmentation is noted in Code block 9. Additional code changes will be needed to support the extra segments. The three dimension eigen calculation can implemented with little effort. This change is expected to drop the execution time by at least half.

11. Machine/System Details

Lenovo ThinkPad
W520

CPU: Intel i7 (Sandy Bridge)
2.4GHz Clk, 64bit, L3 6MB, 4 core, 8 threads

GPU: Nvidia Quadro 2000M (Fermi)
4 SM's, 1.1GHz Clk, warpSize 32, compute capability 2.1
1 mem engine, gMem 2GB, cMem 64KB, 27.5GB/sec

RAM: 4GB
DDR3

SSD: 256GB Vertex4 OCZ

OS:	Ubuntu 12.04 (Linux-x86, 64bit)
Nvidia driver:	304.43
CUDA toolkit:	5.5
GNU GCC:	4.6.3
FSL:	4.1.8
QT4:	4.8.1
VTK:	5.8.0
GSL:	1.9
ITK:	3.20.1
CULA:	Ver R14
DICCCOL code base:	1/30/2012

Abbreviation Glossary

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CULA	CUDA Linear Algebra
DICCCOL	Dense Individualized and Common Connectivity-based Cortical Landmarks
DTI	Diffusion Tensor Image
FSL	FMRIB Software Library
GPU	Graphics Process Unit
GPGPU	General Purpose GPU
HW	Hardware
LAPACK	Linear Algebra Package
MRI	Magnetic Resonant Image
NMR	Nuclear Magnetic Resonance
PCA	Principal Component Analysis
PCIe	PCI Express (Peripheral Component Interconnect Express)

References

[1] Zhu D, Li K, Guo L, Jiang X, Zhang T, Zhang D, Chen H, Deng F, Faraco C, Jin C, Wee C, Yuan Y, Lv P, Yin Y, Hu X, Duan L, Hu X, Han J, Wang L, Shen D, Miller LS, Li L, Liu T. DICCCOL: Dense Individualized and Common Connectivity-based Cortical Landmarks. Cerebral Cortex. 23(4):786-800, 2013.

[2] Lindsay Smith. A tutorial on Principal Components Analysis, 2002

[3] CUDA Linear Algebra Library http://www.culatools.com Release 16a, 2013

[4] Dohrmann. Eigenvalues and eigenvectors of 3 x 3 matrices, 2006

[5] http://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere @ Matt S., 2013

[6] https://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units, 2013

[7] Larsen E, McAllister D. Fast Matrix Multiplies using Graphics Hardware. SC2001 November, 2001

[8] Flucka O, Vetter C, Weina W, Kamena A, Preimb B, Westermannc R. A survey of medical image registration on graphics hardware. computer methods and programs in biomedicine 104 (2011) e45–e57

[9] https://en.wikipedia.org/wiki/BrookGPU, 2013

[10] https://en.wikipedia.org/wiki/CUDA, 2013

[11] Moulika S, Boonna W. The Role of GPU Computing in Medical Image Analysis and Visualization. Proc. of SPIE Vol. 7967, 79670L, 2011

[12] Che S, Boyer M, Meng J, Tarjan D, Jeremy, Sheaffer J, Skadron K. A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 68 (2008) 1370–1380

[13] Guillaume Colin de Verdière. Introduction to GPGPU, a hardware and software background. C. R. Mecanique 339 (2011) 78–89

[14] Sanders J, Kandrot E, CUDA by example—An introduction to general-purpose GPU programming. Addison Wesley, 2010.

[15] Garba M, Gonzalez-Velez H. Asymptotic Peak Utilisation in Heterogeneous Parallel CPU/GPU Pipelines: A Decentralised Queue Monitoring Strategy. Parallel Processing Letters Vol. 22, No. 2 (2012) 1240008

[16] Ryoo S, Rodrigues C, Baghsorkhi S, Stone S, Kirk D, Hwu W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. PPoPP ’08, February 20–23, 2008

[17] Castaño-Díez D, Moser D, Schoenegger A, Pruggnaller S, Frangakis A. Performance evaluation of image processing algorithms on the GPU. Journal of Structural Biology 164 (2008) 153–160

[18] Eklund A, Andersson M, Knutssona H. fMRI analysis on the GPU—Possibilities and challenges. computer methods and programs in biomedicine 105 (2012) 145–161

[19] Otten R, Vilanova A, van de Wetering H. Illustrative White Matter Fiber Bundles. Eurographics/ IEEE-VGTC Symposium on Visualization 2010. Volume 29 (2010), Number

[20] MRI registration: Accelerating image registration of MRI by GPU-based parallel computation
[21] fMRI permutations: Fast Random Permutation Tests Enable Objective Evaluation of Methods for Single-Subject fMRI Analysis
[22] Zhu F, Gonzalez D, Carpenter T, Atkinsona M, Wardlawb J. Parallel perfusion imaging processing using GPGPU. computer methods and programs in biomedicine 108 (2012) 1012–1021
[23] Hernandez M, Guerrero G, Cecilia J, Garcıa J, Inuggi A, Jbabdi S, Behrens T, Sotiropoulos S. Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs. PLoS ONE 8(4): e61892. doi:10.1371/journal.pone.0061892 (2013)
[24] http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FSL
[25] http://fsl.fmrib.ox.ac.uk/fsl/fsl4.0/fdt/fdt_bedpostx.html
[26] Lee J, Kim D. Divide et Impera: Acceleration of DTI Tractography Using Multi-GPU Parallel Processing. Imaging Syst Technol, 23, 256–264, 2013
[27] Wang Y, Du H, Xia M, Ren L, Xu M, Xie T, Gong G, Xu N, Yang H, He Y. A Hybrid CPU-GPU Accelerated Framework for Fast Mapping of High-Resolution Human Brain Connectome. PLoS ONE 8(5): e62789. doi:10.1371/journal.pone.0062789 (2013)
[28] class tictoc{} in tictoc.hpp from Ofir Pele (2010)
[29] https://en.wikipedia.org/wiki/Brodmann_area (2013)
[30] https://en.wikipedia.org/wiki/Cerebral_cortex (2013)
[31] https://en.wikipedia.org/wiki/File:Human_brain_right_dissected_lateral_view_description.JPG (2013)
[32] https://en.wikipedia.org/wiki/White_matter (2013)
[33] Stieltjes B, Brunner R, Fritzsche K, Laun F. Diffusion Tensor Imaging Introduction and Atlas. Springer-Verlag Berlin Heidelberg
[34] Faro S, Mohamed F, Law M, Ulmer J. Functional Neuroradiology Principles and Clinical Applications. Springer Science+Business Media, LLC 2011
[35] https://www.youtube.com/watch?v=djAxjtN_7VE (Nov 2013)
[36] Beaulieu C. The basis of anisotropic water diffusion in the nervous system – a technical review. NMR Biomed. 2002;15(7–8):435–55.
[37] Coenen V, Schlaepfer T, Allert N, Mädler B. Diffusion Tensor Imaging and Neuromodulation: DTI as Key Technology for Deep Brain Stimulation. International Review of Neurobiology, Volume 107: 207-234 (2012)
[38] https://en.wikipedia.org/wiki/Tensor
[39] https://en.wikipedia.org/wiki/Sphere
[40] https://en.wikipedia.org/wiki/Ellipsoid
[41] https://en.wikipedia.org/wiki/File:DTI-sagittal-fibers.jpg
[42] http://fsl.fmrib.ox.ac.uk/fsl/fsl4.0/flirt/overview.html
[43] https://en.wikipedia.org/wiki/Intel_MIC

[44] https://en.wikipedia.org/wiki/Unified_atomic_mass_unit
[45] Hanbo Chen, Yu Zhao, Tuo Zhang, Hongmiao Zhang, Hui Kuang, Meng Li, Joe Z. Tsien, Tianming Liu, Construct and Assess Multimodal Mouse Brain Connectomes via Joint Modeling of Multi-scale DTI and Neuron Tracer Data, accepted, MICCAI 2014.

[i] Color information was missing from the original data set. RGB color information was added. The fiber shape and pathways remain unaltered.