UNIVERSIDAD DE GUANAJUATO

UNIVERSITY OF GUANAJUATO

NATURAL AND EXACT SCIENCES DIVISION
MATHEMATICS DEPARTMENT
CAMPUS GUANAJUATO

PARALLELIZATION OF COMMIT USING CUDA

Acceleration with GPU of a large-scale problem for
microstructure informed tractography

PROFESSIONAL THESIS

A THESIS PRESENTED FOR THE DEGREE OF:
LICENCIATURA EN COMPUTACIÓN

BY:
ERICK HERNÁNDEZ GUTIÉRREZ

CO-ADVISORS:
DR. ALONSO RAMÍREZ MANZANARES
DR. JOSÉ LUIS MARROQUÍN ZALETA

EXTERNAL ADVISOR:
DR. ALESSANDRO DADUCCI

GUANAJUATO, GUANAJUATO. **SEPTEMBER 2018**

ABSTRACT

This document presents an optimization of the COMMIT framework developed by the Dr. Alessandro Daducci and his collaborators. COMMIT framework has been used to filter tractograms which are useful to study the connections in the human brain.

We parallelized with the CUDA language the algebraic operations Ax and $A^t y$ in order to accelerate the optimization procedure necessary to filter a tractogram with COMMIT. The results of our parallel implementation of the operations were validated by comparing the results with the current version of COMMIT. This work shows experiments with real human brain data which demostrate that the parallel versions of the operations Ax and $A^t y$ significantly reduced the computational time required to filter a tractogram.

This thesis contribute with a faster version of the COMMIT framework which uses a NVIDIA GPU to accelerate the operations Ax and $A^t y$ along with backward compatibility with the previous COMMIT scripts.

CONTENTS

1 Introduction **1**

1.1 Diffusion MRI . . . 1

1.2 Tractogram . . . 2

1.3 Tissue Models . . . 3

1.4 COMMIT Framework . . . 4

1.5 Motivation . . . 5

2 The CUDA Programming Model **6**

2.1 General Background . . . 6

2.1.1 Branch Predictor . . . 6

2.1.2 Memory RAM DDR/GDDR . . . 7

2.2 Program Structure . . . 7

2.3 Kernels . . . 7

2.4 Thread Hierarchy . . . 8

2.4.1 CUDA Grid . . . 8

2.4.2 CUDA Block . . . 8

2.4.3 CUDA Wrap . . . 9

2.4.4 CUDA Thread . . . 9

2.5 Device Memory . . . 11

2.5.1 Registers . . . 12

2.5.2 Local Memory . . . 12

2.5.3 Shared Memory . . . 13

2.5.4 Global Memory . . . 13

2.5.5 Constant Memory . . . 14

2.6 Measuring Performance . . . 15

3 COMMIT and Problem Statement **17**

3.1 Candidate Tracts . . . 17

3.2 Response Functions . . . 17

3.3 Formulation . . . 19

3.4 Fitting . . . 20

4 Methods and Development **21**

4.1 Look-up Table with the Precomputed DW-MR Signals . . . 21

4.2 Sparse Structure . . . 22

4.2.1 Block Matrix IC . . . 22

4.2.2 Block Matrix EC . . . 24

4.2.3 Block Matrix ISO . . . 24

4.3 Preprocessing . . . 24

4.4 Operation Ax . . . 28

4.4.1 Algorithm IC . . . 29

4.4.2 Algorithm EC . . . 30

4.4.3 Algorithm ISO . . . 32
4.5 Operation $A^t y$ . . . 32
4.5.1 Algorithm IC . . . 33
4.5.2 Algorithm EC . . . 38
4.5.3 Algorithm ISO . . . 40
4.6 Improving the Speed of the Ax operation . . . 41
4.7 Integration with COMMIT . . . 43

5 Experiments and Results **49**
5.1 Operation Ax . . . 49
5.2 Operation $A^t y$ . . . 52
5.3 Model Fitting . . . 53

6 Conclusions and Future Work **55**
6.1 Conclusions . . . 55
6.2 Areas of Improvement and Future Work . . . 56

A Compartment Models **58**
A.1 Intra-cellular Models . . . 58

LIST OF FIGURES

1.1 2D visualization of a dMRI image. Image taken from www.valleyradiologync.com. . 2

1.2 Structure of an axon. Image taken from https://qbi.uq.edu.au/brain/brain-anatomy/axons-cable-transmission-neurons. 2

1.3 Diffusion MRI image paired with a 2D visualization of its DTI tractography. Image taken from MGH Martinos Center. 3

1.4 Left: Two-dimensional image I with nine voxels and three intra-cellular compartments (fibers). Right: The same image I with fibers modeled with the cylinder model, see appendix. 4

1.5 Block diagram of top-down strategies to combine tractography reconstructions with local properties of the tissue. Image taken from [3.1]. 4

1.6 The COMMIT model. Image taken from [3.1]. 5

1.7 Top: Ground-truth image from the FiberCup data showing a tractogram obtained with the global tractography algorithm GIBBS. Bottom: Filtered tractogram obtained with COMMIT. Image taken from [3.1]. 5

2.1 Simple CUDA Grid. 11

2.2 Small example of a full CUDA programming model and memory hierarchy. 12

3.1 2D image with 4 voxels. Green color is used to indicate which voxels are intersected by the fiber. The fiber is modeled by using the cylinder model and 4 response functions. 18

3.2 Cylinder response function transformed with the operators T_v^f and E_v^f. 18

3.3 Example of extra-cellular compartments in a voxel v. 19

4.1 Example of a matrix A^{IC}. Blank spaces are zero signals. Cylinder response functions are used. 23

4.2 Example of the COO format for the storage of sparse matrices. 23

4.3 Distribution of the threads in the algorithm for $A^{IC}x$. 29

4.4 Distribution of CUDA blocks and CUDA threads in the computation of A^{tIC}. 34

4.5 Steps of a parallel reduction. Each orange circle indicates an operation performed by a thread. This image is property of NVIDIA Corporation. 35

4.6 Diagram of a simple fitting process using COMMIT framework. Curly brackets show some of the process inside blocks. 44

4.7 Block diagram showing a fitting using the modified COMMIT. Curly brackets show some of the process inside each block. 46

5.1 The computational time required by the operation Ax using 1, 2, 4, 8, 16 and 32 CPU threads. Horizontal axis represents the number of CPU threads and vertical axis represents the average time in seconds of the 50 experiments. The blue vertical lines represent the variance in the 50 experiments. The red line shows the time of the GPU version. Green line shows the GPU version with the modified IC algorithm introduced in the Section 4.6. 51

5.2 Required computational time to compute the operation Ax in CPU by varying the sparsity index of the vector x. 52

5.3 The computational time required by the operation A^ty using 1, 2, 4, 8, 16 and 32 CPU threads. Horizontal axis represents the number of CPU threads and vertical axis represents the average time of 50 experiments in seconds. The blue vertical lines represent the variance of the 50 experiments. The green line shows the time of our GPU implementation in CUDA. 53

6.1 Threads accessing near addresses in a 2D texture memory. This image is property of NVIDIA Corporation. 56

6.2 Histogram of the number of fiber segments per voxel. 57

LIST OF TABLES

2.1 CUDA functions qualifiers. 8

5.1 Average times of the experiments. 51

5.2 Relative error between vectors y_{gpu} and y_{cpu}. The vector y_{gpu} represents the output vector obtained with the GPU version of the operation Ax and y_{cpu} represents the output vector obtained with CPU version of Ax. 51

5.3 Relative error between vectors x_{gpu} and x_{cpu}. The vector x_{gpu} represents the output vector obtained with the GPU version of the operation $A^t y$ and x_{cpu} represents the output vector obtained with CPU version of $A^t y$. 53

5.4 Computational time required to perform the model fitting with 1 CPU thread versus our implementations in GPU with CUDA. 54

CHAPTER 1

INTRODUCTION

1.1 Diffusion MRI

Diffusion Magnetic Resonance Imaging (Diffusion MRI or dMRI) [1.4] is a radiologic method in medical image processing and neuroscience. It provides information of the internal connections in a brain and it is a non-invasive way. In diffusion MRI, image contrast is determined by the random microscopic motion of water [1.3] produced by the thermal motion on the water molecules. Diffusion MRI data can be presented by using 2D and 3D visualizations [1.5], see Figure 1.1.

A *pixel* is the smallest unit when dividing a 2D space into discrete [1.10] regions. It is very common that all the pixels of one image have the same size, i.e. the same width and height. Each pixel can be accessed by using two integer values, one for each space component of the image. This concept of pixel can be extended to 3D spaces. In the case of 3D spaces pixels become *voxels*. So, a voxel is the smallest unit of volume when dividing a 3D space into discrete regions [1.10].

Let $I \in \mathbb{R}^{n_x \times n_y \times n_z \times n_s}$ be a dMRI image where n_x, n_y and n_z are the number of voxels per dimension in the image. Our image I is composed of n_s samples acquired over the $n_v = n_x n_y n_z$ voxels. In this document, we will not discuss the science behind the measured dMRI signal because it goes beyond this document. We just restrict ourselves to receive I as input.

Figure 1.1: 2D visualization of a dMRI image. Image taken from www.valleyradiologync.com.

1.2 Tractogram

An *axon*, or nerve fiber, is a long slender projection of a nerve cell (neuron), that conducts electrical impulses away from the neurons's cell body or soma [1.11], see Figure 1.2. Axons are the principal line transmission for the nervous system as they work as "cables" and they are grouped in bundles.

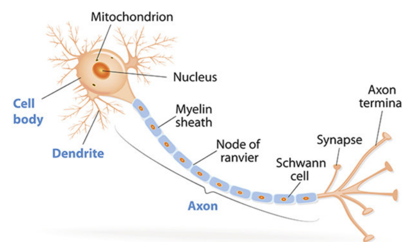

Figure 1.2: Structure of an axon. Image taken from https://qbi.uq.edu.au/brain/brain-anatomy/axons-cable-transmission-neurons.

Diffusion MRI data allows to aproximate the main paths of large sets of axons with polylines called streamlines or fibers [1.1]. The set containing all the fibers is commonly called tractogram and it represents the essence of the conections inside the brain. Tractograms can be visualized as two- or three-dimensional images, see Figure 1.3.

Let $\mathcal{F}$ be a tractogram calculated from our dMRI image I. There are plenty of tractography algorithms to calculate $\mathcal{F}$ and some of them can be combined to get a better estimation of the fibers in $\mathcal{F}$, for instance [1.7, 1.8, 1.2]. However, this document will not focus in the estimation of the fibers in $\mathcal{F}$. Instead, we will assume that $\mathcal{F}$ is given as input.

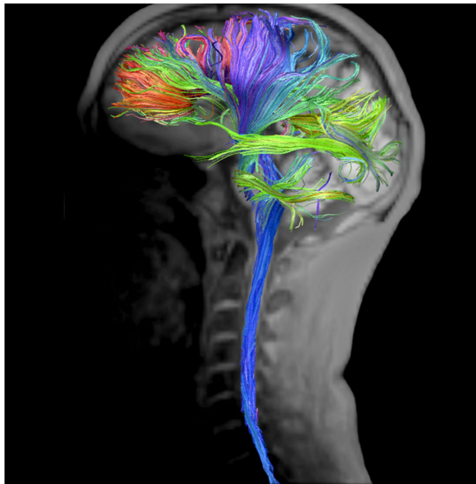

Figure 1.3: Diffusion MRI image paired with a 2D visualization of its DTI tractography. Image taken from MGH Martinos Center.

1.3 Tissue Models

We shall see in the next chapters that the dMRI signal have to be proceeded in order to compute brain structure features at each voxel in I. To do that we need a tissue model. COMMIT models the brain tissue with a model based on [3.2] which combines three different compartments:

1. Intra-cellular (IC): This compartment refers to the signal coming from water inside the axons.
2. Extra-cellular (EC): This compatment refers to the signal coming from water outside the axons.
3. Isotropic (ISO): This compartmen refers to the signal coming from compartments without cellular structures

The models used for each compartment are described in the appendix A of this document. The contribution signals obtained from these models are called $response\ functions$. Figure 1.4 shows a visual modeling example of the intra-cellular compartments (fibers) by using cylinder response functions, see [3.1].

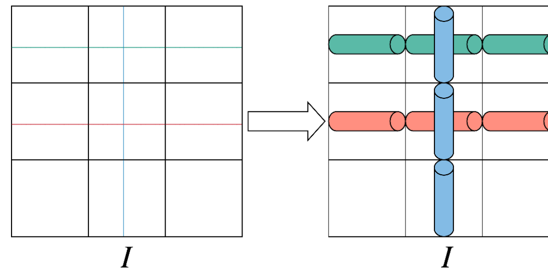

Figure 1.4: Left: Two-dimensional image I with nine voxels and three intra-cellular compartments (fibers). Right: The same image I with fibers modeled with the cylinder model, see appendix.

1.4 COMMIT Framework

Even with the best tractography algorithms, you will never get a perfect tractogram $\mathcal{F}$, i.e. there will be artefactual tracts in $\mathcal{F}$ that do not represent an anatomical reality (*false positives*) [1.9]. This becomes a big challenge because tractograms are central to the study of human brain connectivity. In contrast to the large number of tractography algorithms, there has been just a few algorithms on false positives detection [1.6].

Convex Optimization Modeling for Micro-structure Informed Tracttography (COMMIT) is a framework originally developed in MATLAB, which detects and removes false positives in a tractogram. Nowadays, COMMIT is implemented as a python module and it is available in [4.2]. Taking as inputs the tractogram $\mathcal{F}$, dMRI image I and local brain tissue models, COMMIT performs a global convex optimization procedure to calculate a filtered tractogram $\widetilde{\mathcal{F}} \subseteq \mathcal{F}$ by removing potential false positives fibers from $\mathcal{F}$, see Figure 1.5.

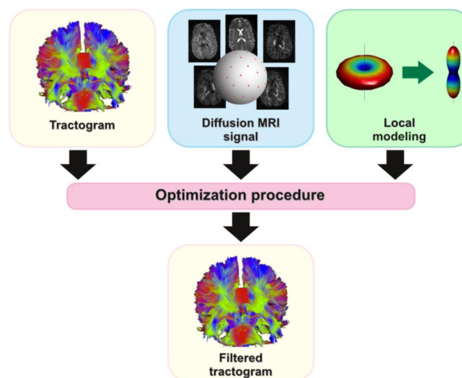

Figure 1.5: Block diagram of top-down strategies to combine tractography reconstructions with local properties of the tissue. Image taken from [3.1].

COMMIT builds a dictionary A by using $\mathcal{F}$ and the brain tissue model and flattens the image I

into a signal vector y. Then, it formulates and solves for x the equation showed in Figure 1.6. We shall introduce a deeper explanation of this formula in chapter 3.

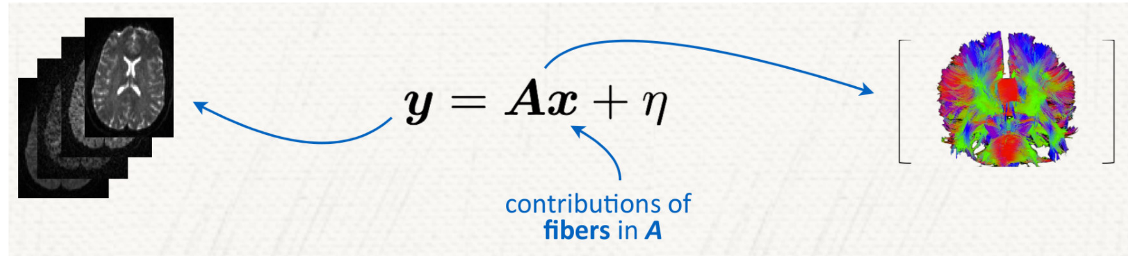

Figure 1.6: The COMMIT model. Image taken from [3.1].

COMMIT has demonstrated its effectiveness as it is shown in the experiments in [3.1] and in the Figure 1.7. It has a low memory usage due to optimized sparse structures, see Chapter 4. COMMIT can run in a mid-range laptop even with real full-brain tractograms.

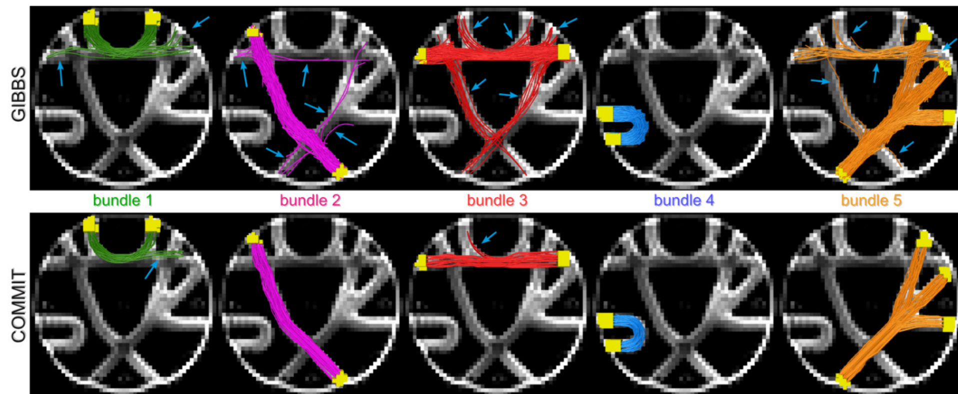

Figure 1.7: Top: Ground-truth image from the FiberCup data showing a tractogram obtained with the global tractography algorithm GIBBS. Bottom: Filtered tractogram obtained with COMMIT. Image taken from [3.1].

1.5 Motivation

For big data sets, the optimization process of COMMIT can take days of intense computational time even with a super computer [3.1]. The bottleneck lies in the computation of the operations Ax and $A^t y$ several times during the optimization procedure. Even though these are implemented by using the programming language C, the required time for a simple operation Ax and $A^t y$ is still a lot because of the large size of the dictionary A. In this thesis we introduce a parallelized version of the operations Ax and $A^t y$ by using CUDA programming language. Our target is to reduce the amount of time necessary to filter the tractogram $\mathcal{F}$ keeping the flexibility of the COMMIT framework.

CHAPTER 2

THE CUDA PROGRAMMING MODEL

The main target of this document is to speed up the COMMIT framework by using a NVIDIA Graphics Processor Unit (NVIDIA GPU) with the CUDA computing platform. In this chapter we are going to discuss some general GPU facts along with CUDA structure, thread hierarchy and device memory.

2.1 General Background

2.1.1 Branch Predictor

When you design a parallel algorithm for GPUs, it is very important to be sure that all threads are executing the same instruction at the same time, otherwise, there will be a dramatically reduction of performance in the program. This happens because GPUs, unlike CPUs, do not have branch predictor. This predictor predicts at each conditional check like if, *else*, for and *while* and it speculates which path the thread is going to take, then the predictor loads the corresponding instructions. If the thread goes in a different way, the predictor just flushes all the preloaded instructions and the process starts again [2.9].

The modern x86 CPUs include complex branch predictor hardware architecture just to take care of a few amount of threads. So, it would be impossible to fit brach predictor hardware to monitor the large number of threads inside the GPU. Moreover, GPUs do not have branch predictor because most efficient threads in the GPU programs do not have branches.

2.1.2 Memory RAM DDR/GDDR

Another, significant aspect of designing algorithms for GPUs is the memory access. The RAM memory for GPU is designed different from the CPU one. Both Double Data Rate (DDR) and Graphic Double Data Rate (GDDR) memories are based on SDRAM (Synchronous Dynamic Random Access Memory) chips [2.8]. But the first one (used in CPUs) focuses on quickly providing small data blocks, while the second one (used in GPUs) slowly provides large data blocks. DDR memory type is more useful for CPUs because CPUs have a few amount of fast threads that can process small data blocks quickly. However, GPUs have a large amount of slow threads, so it is better to get large data blocks from memory and to process them in parallel. Consequently, this changes how we design algorithms for GPUs.

2.2 Program Structure

CUDA is a general-purpose parallel computing platform and programming model. Using CUDA, you can access to NVIDIA GPUs for parallel computation in a very similar way that CPUs. The CUDA platform is composed of CUDA-accelerated libraries, compiler directives, application programming interfaces and extensions to standard programming languages like C, C++, Fortran and Python. This document uses only CUDA C/C++ programming. CUDA C/C++ is an extension of standard ANSI C/C++ with some language extensions to manage threads, memory and others tasks. CUDA is also scalable to several devices.

CUDA programs are compoosed of two parts: *host* code and *device* code. Host code runs on the CPU and device code runs on GPU. CUDA has its own compiler called nvcc which is based on the widely LLVM open source compiler infrastructure [2.6]. CUDA nvcc compiler separates the device and the host code during compilation process. Host code is sent to a traditional C/C++ compiler like gcc/g++ or Microsoft Visual C/C++ and device code is further compiled by nvcc and mapped to the GPU.

2.3 Kernels

An important component of the CUDA programming model is the *kernel*. Functions that are executed on a NVIDIA GPU are called kernels. As we saw in a previous sub-section, all CUDA programs are composed of GPU and CPU code. So, there is a mechanism to differentiate between CPU and GPU functions. NVIDIA extended C/C++ by adding qualifiers *__global__*, *__device__* and *__host__* in order to support mixed code. A simple explanation of the functionality of these qualifiers is showed in Table 2.1.

	Called from the	Executed on the
__global__	CPU	GPU
__device__	GPU	GPU
__host__	CPU	CPU

Table 2.1: CUDA functions qualifiers.

There is nothing special about __device__ and __host__. In fact, qualifier __host__ can be omitted most of the time unless you want to combine __device__ and __host__ to instruct compiler to generate one version of the same function for GPU and another for CPU.

Calling a function with __global__ requires to add extra information. When a __global__ function is called from CPU, it is necessary to include angle brackets syntax <<<...>>> and a numeric tuple. These two numbers influence how the runtime will launch device code. This couple of numbers tells compiler how many CUDA blocks and how many CUDA threads per block will be used to launch that kernel. In the next sub-section we will explain more about CUDA blocks and threads.

2.4 Thread Hierarchy

2.4.1 CUDA Grid

At the top level of the hierarchy, a CUDA grid is organized as an array of CUDA blocks. This array can be single or two dimensional. The number of dimensions and the number of CUDA blocks in each dimension is specified by the programmer at the kernel launch, with a limit of 2^{16} CUDA blocks per dimension. When we launch a kernel, a CUDA grid is assigned to the kernel. There is only one single CUDA grid per kernel and dimensions of the CUDA grid cannot be changed during execution.

2.4.2 CUDA Block

Each CUDA block is organized into an array of CUDA threads. As well as CUDA grids, programmers can control the block dimension and the number of threads per block. All blocks of a grid have the same dimension. The total number of threads is limited at 1024 threads. For example, a block with dimensions (512, 1, 1), (32, 32) and (100) are allowed, but (512, 512) will not be allowed because it exceeds the 1024 threads limit.

2.4.3 CUDA Wrap

Inside each block its threads are grouped in groups of 32 threads called *wrap*. Wrap size is fixed and programmers do not have directly control of the wraps. The GPU loads one instruction for every wrap and all the threads in that wrap should execute the same instruction. If one or more threads in a wrap execute different instructions. The GPU first loads one instruction for some of the threads in that wrap, then it loads another instruction for the rest of the threads in that wrap. This phenomenon is called *thread divergence* and it damages performance because some threads in a wrap wait for others.

2.4.4 CUDA Thread

A thread is the smallest sequence of programmed instruction that can be managed indepently by the GPU. So, if we launch a kernel, each line in the kernel is going to be independently executed by all threads in the grid. As an example, let us suppose that we want to launch a kernel $example()$ with a (2, 3) grid where each block is (3, 4) size. The code showed down below shows the complete code in CUDA.

```
1  #include <stdio.h>
2  #include <cuda.h>
3
4  __global__ void example(){
5      unsigned int bidx = blockIdx.x;
6      unsigned int bidy = blockIdx.y;
7      unsigned int tidx = threadIdx.x;
8      unsigned int tidy = threadIdx.y;
9
10     // each thread prints its own id inside its block
11     printf("block  id = (%d, %d)\n", bidx, bidy);
12     printf("thread id = (%d, %d)\n", tidx, tidy);
13 }
14
15 int main()
16 {
17     // define the size of the grid
18     dim3 gridsize(2, 3);
19
20     // define the size of each block
21     dim3 blocksize(3, 4);
22
23     // launch kernel in the GPU from the CPU
24     example<<<gridsize, blocksize>>>();
25
26     return 0;
27 }
```

We can note that we are using $2 * 3 = 6$ blocks each one with $3 * 4 = 12$ threads. So, we have 72 threads running in parallel the function $example()$. You can also see that inside the function each thread uses global variables $blockIdx$ and $threadIdx$ to print its own id and the block id. In this case, we have 2D dimensional ids because we are using 2D grids and 2D blocks. An illustrative explanation of the structure inside CUDA of this example is showed in the Figure 2.1.

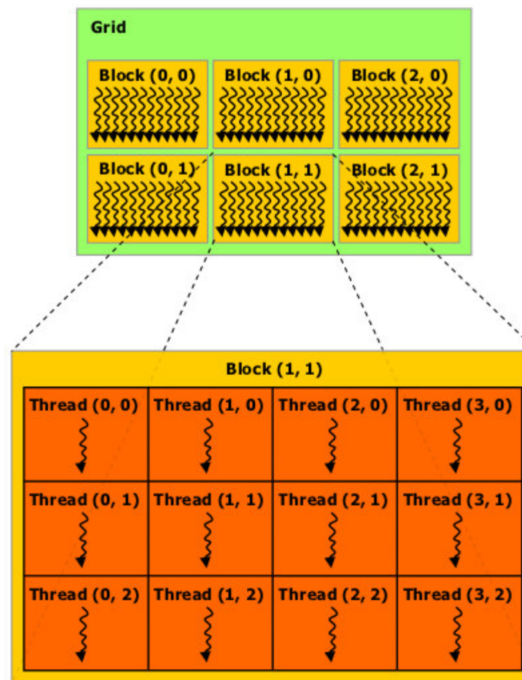

Figure 2.1: Simple CUDA Grid.

2.5 Device Memory

In contrast to CPU, CUDA programming model has several types of memory. Correct memory access and managment has an important impact on performance. The CUDA memory model can be visualized in Figure 2.2.

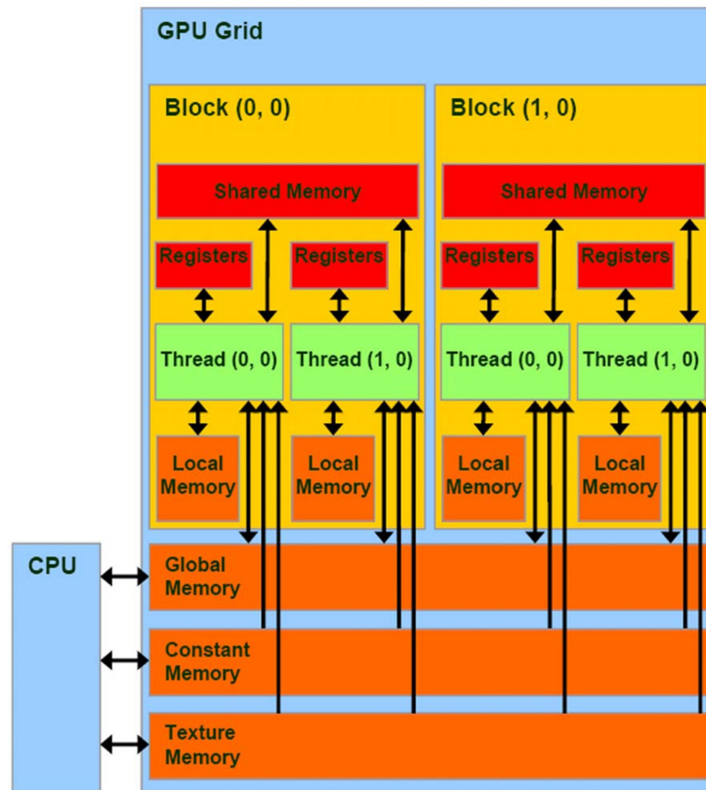

Figure 2.2: Small example of a full CUDA programming model and memory hierarchy.

Each type of memory will be described in the following sub-sections.

2.5.1 Registers

Registers are the fastest type of memory in CUDA. Small arrays and automatic variables are also stored in registers. Each thread has its own registers and these registers cannot be accessed by other threads. There is a limit of 255 registers per thread on lastest NVIDIA GPU architectures like Kepler, Maxwell and Pascal.

2.5.2 Local Memory

Every thread in a block has its own local memory space. However, The name "Local Memory" is confusing because variables stored in local memory are in the same physical location as global memory. So, if you access to local memory you achieve high latency and low bandwidth accesses. When a variable in a kernel does not fit into the register memory, it is allocated into local memory which is slower than the registers. Structures and arrays that use a lot of memory are also spilled to local memory.

2.5.3 Shared Memory

This type of memory is similar to CPU L1 cache, moreover, shared memory is also configurable. Shared memory has lower latency and higher bandwidth than global memory because it is on-chip. However, each block has a limited amount of shared memory. Each block has its own shared memory space and that memory is visible and shared between threads inside the same block. In fact, this memory is the main tool for thread cooperation between threads in the same block. Limit of shared memory per block depends on the GPU. As an example, the NVIDIA Quadro P6000, which is used later in experiments, has 49152 bytes per block. Variables that are going to be allocated in shared memory must use $__shared__$ identificator and they must be declared in kernel code.

2.5.4 Global Memory

We had already heard about global memory in the last paragraphs. This is the largest, but also the slower, type of memory available in CUDA. This memory physically resides in the GDDR chips around the GPU chipset. Every thread in the GPU can write to or read from the global memory. That is why this type of memory is the most used.

Global memory must be managed, released and transferred by the CPU. To allocate static global memory, you need to add the $__device__$ identifier before declaration. Dynamic allocation and release can be performed through the functions $cudaMalloc()$ and $cudaFree()$, which are very similar to the standard CPU $malloc()$ and $free()$ functions. Finally, to transfer global memory we have to use the $cudaMemcpy()$ function. An example of memory managment is showed below.

```
1  void main()
2  {
3       //allocate dynamic array in CPU
4       int* cpu_array;
5       cpu_array = (int*)malloc(SIZE*sizeof(int));
6
7       //allocate dynamic array in GPU
8       int* gpu_array;
9       cudaMalloc((void**)&gpu_array, SIZE*sizeof(int));
10
11      //transfer data from CPU to GPU
12      cudaMemcpy(gpu_array, cpu_array, SIZE*sizeof(int),
            cudaMemcpyHostToDevice);
13
14      //launch kernel to process data
15      kernel<<<NUM_BLOCKS, NUM_THREADS>>>(gpu_array);
16
17      //transfer processed data back to CPU
18      cudaMemcpy(cpu_array, gpu_array, SIZE*sizeof(int),
            cudaMemcpyDeviceToHost);
19
20      //release dynamic array
21      cudaFree(gpu_array);
22
23      //release dynamic array
24      free(cpu_array);
25 }
```

There is not distinction between pointers for CPU and pointers for GPU. CUDA devices can share a unified address space with the CPU [2.4]. So, you do not have to add any identifier to the pointers. But pointers pointing to memory allocated in GPU can only be accessed from kernels by passing pointers as parameters in the function arguments. There are more functions to manage this memory, for a deep understanding of all these concepts we refer the reader to the following manuscripts [2.5, 2.6, 2.7].

2.5.5 Constant Memory

Constant memory resides in device memory and it is cached in a particular constant cache [2.6]. If all threads in a wrap read a variable from the same memory address of constant memory, that data will be read as fast as if they were read from registers. However, If some threads access different

addresses the data is serialized, so performance is affected. Threads cannot write data to constant memory. So, CPU must initialize constant memory using *cudaMemcpyToSymbol*(). Variables that are going to be allocated in constant memory must be in a global scope and they must use the __*constant*__ identifier. Constant variables do not need to be passed as parameters. For example:

```
1  //constant variable in GPU
2  __constant__ int GPU_M;
3
4  //constant variable in CPU
5  const int M;
6
7  void main()
8  {
9      //initialize in CPU
10     M = 10;
11
12     //initialize in GPU
13     cudaMemcpyToSymbol(GPU_M, &M, sizeof(int));
14
15     //launch kernel
16     kernel<<<NUM_BLOCKS, NUM_THREADS>>>();
17 }
```

2.6 Measuring Performance

Time comparison is a very important part of this work. There are several ways to measure kernel performance. The easier way is to use a CPU timer to measure kernel execution from the host side. But it is not as precise as CUDA-specific timing routines. The CUDA API provides functions and structures that allow to insert events at any point in a stream. This CUDA events can be used for more things than just measuring time. But we will use events just for performance measurement.

To measure time we are going to need two events. One will be set at the beginning of the kernel launch and the second one at the end of the kernel launch. Events have to be declared and created from host side. You can see an example code down below:

```
int main()
{
    // declare events
    cudaEvent_t start;
    cudaEvent_t stop;

    // create and initialize events
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // set one event before and after lauching
    cudaEventRecord(start);
    kernel<<<NUM_BLOCKS, NUM_THREADS>>>(arguments);
    cudaEventRecord(stop);

    // calculate elapsed time
    float elapsed;
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsed, start, stop);

    //print elapsed time
    print("elapsed time = %f ms/n", elapsed);
}
```

CHAPTER 3

COMMIT AND PROBLEM STATEMENT

3.1 Candidate Tracts

COMMIT is a very flexible framework because it accepts and works with every tractogram [4.2, 3.1]. Hence, the tractogram $\mathcal{F}$ can be calculated from I by using any tractography algorithm. However, COMMIT does not add fibers to $\mathcal{F}$. For this reason, the only requisite is that $\mathcal{F}$ contains fibers that connect pairs of actually connected brain regions by following the right brain pathways. These fibers are called *true positives*.

3.2 Response Functions

Before considering the formulation of COMMIT it is important to talk about the $response\ functions$. These functions are used to map the dMRI signal contribution of each tract in $\mathcal{F}$ at every voxel of the image I [3.1]. In the COMMIT framework, the magnetic resonance signal is acquired along n_s 3D orientations. The number n_s of direction samples is important because one response function $R \in \mathbb{R}^{n_s}$ carries the response signal of one compartment across the n_s directions samples. Usually, we use more than one response function per comparment [3.1]. Let n_a, n_b and n_c be the number of response functions for the IC, EC and ISO comparments respectively. Figure 3.1 ilustrate an example of an image with a fiber intersecting two voxels. This fiber is modeled by using four different response functions. Each one of these response functions represents the intra-cellular microstructure with cylinders with a different diameter, see Apendix A.

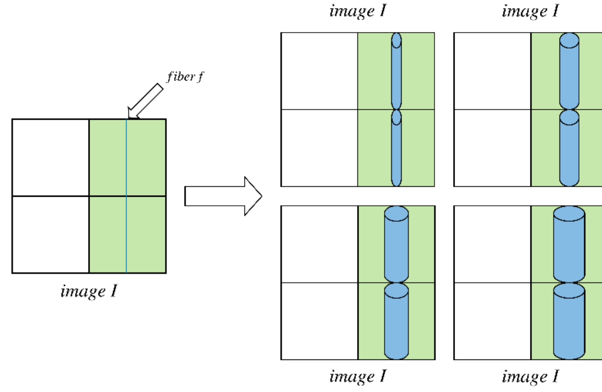

Figure 3.1: 2D image with 4 voxels. Green color is used to indicate which voxels are intersected by the fiber. The fiber is modeled by using the cylinder model and 4 response functions.

We shall introduce the operators $T_v^f : \mathbb{R}^{n_s} \to \mathbb{R}^{n_s}$ and $E_v^f : \mathbb{R}^{n_s} \to \mathbb{R}^{n_s}$. Operator T_v^f rotates a response function R to match the local orientation of the fiber $f \in \mathcal{F}$ in the voxel $v \in \{0, \ldots, n_v - 1\}$ and E_v^f scales R by the actual length in mm of f in the voxel v, see Figure 3.2.

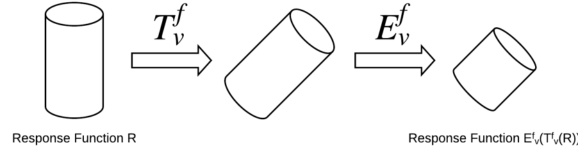

Figure 3.2: Cylinder response function transformed with the operators T_v^f and E_v^f.

On the other hand, let $S_v \in \mathbb{R}^{n_s}$ be the predicted signal carrying the total contribution of the tracts in $\mathcal{F}$ to the voxel v. We normally need very expensive computational nonlinear procedures to calculate these signals. Fortunately, there are linear approaches that can estimate these compartments quickly [3.7, 3.8]. Thus, given a voxel v in I, we can write S_v as a linear combination with the following formula:

$$S_v = S_v^{IC} + S_v^{EC} + S_v^{ISO} = \sum_{f \in \mathcal{F}} \sum_{j=0}^{n_a - 1} x_j^f E_v^f \left(T_v^f \left(R_j^{IC} \right) \right) + \sum_{f \in \mathcal{F}} \sum_{j=0}^{n_b - 1} x_j^f T_v^f \left(R_j^{EC} \right) + \sum_{j=0}^{n_c - 1} x_j^v R_j^{ISO}. \quad (3.1)$$

We should also consider that in a voxel v there are always a lot of fibers with the same orientation. So, we do not need to calculate the extra-cellular signal for all those fibers. We can replace a fiber bundle by an extra-cellular compartment with the same orientation, see Figure 3.3.

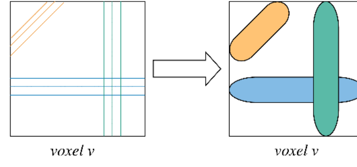

Figure 3.3: Example of extra-cellular compartments in a voxel v.

Thus, equation 3.1 can be simplified by calculating the sets $\mathcal{O}_v$ for every voxel $v \in \{0, \ldots, n_v - 1\}$. The set $\mathcal{O}_v$ represents the set of all main orientations in the voxel v. The total number of extra-cellular compartments is denoted with n_e, so we have

$$n_e = \left| \bigcup_{v=0}^{n_v-1} \mathcal{O}_v \right|. \tag{3.2}$$

Therefore, equation 3.1 can be written as:

$$S_v = S_v^{IC} + S_v^{EC} + S_v^{ISO} = \sum_{f \in \mathcal{F}} \sum_{j=0}^{n_a-1} x_j^f E_v^f \left(T_v^f \left(R_j^{IC} \right) \right) + \sum_{e \in \mathcal{O}_v} \sum_{j=0}^{n_b-1} x_j^e T_v^e \left(R_j^{EC} \right) + \sum_{j=0}^{n_c-1} x_j^v R_j^{ISO} \tag{3.3}$$

where $R_j^{IC} \in \mathbb{R}^{n_s}$, $R_j^{EC} \in \mathbb{R}^{n_s}$ and $R_j^{ISO} \in \mathbb{R}^{n_s}$ are the jth intra-cellular, extra-celluar and isotropic response functions respectively and coefficients x_j^f, x_j^e and x_j^v represents the global contribution of the MR signal from each compartment within the voxel. We will denote these compartmental signals as base functions.

3.3 Formulation

The dMRI image I and the tractogram $\mathcal{F}$ can be modeled with an operator $\mathcal{L} : \mathcal{F} \to I$ such that

$$I = \mathcal{L}(\mathcal{F}) + \eta. \tag{3.4}$$

Then $\mathcal{L}$ models the signal contribution of each fiber in all acquired voxels and η is the acquisition noise [3.1]. Now, the mapping $\mathcal{L} : \mathcal{F} \to I$ can be implemented as a linear operator by rewriting equation 3.1 as a matrix by using all the signals S_v. Let A be the $n_v n_s \times m$ matrix of the operator $\mathcal{L}$ where $m = n_f n_a + n_e n_b + n_v n_c$. Let $y \in \mathbb{R}_+^{n_v n_s}$ be the vectorized (flattened) version of the dMRI image I. Let $x \in \mathbb{R}_+^m$ be the vector standing for the contributions of the m basis functions in A. Then equation 3.4 can be written as:

$$y = Ax + \eta. \tag{3.5}$$

The matrix A is a block matrix in the form:

$$A = \left[\, A^{IC} \mid A^{EC} \mid A^{ISO} \,\right] \tag{3.6}$$

where $A^{IC} \in \mathbb{R}_+^{n_v n_s \times n_f n_a}$, $A^{EC} \in \mathbb{R}_+^{n_v n_s \times n_e n_b}$ and $A^{ISO} \in \mathbb{R}_+^{n_v n_s \times n_v n_c}$. All matrices encapsulate, respectively, intra-cellular, extra-cellular and isotropic contributions to the image. In the next chapter, we will explain in depth the structure of these matrices. Note that if we calculate S_v for all the voxels $v \in \{0, \ldots, n_v - 1\}$ in the image I, we will have calculated the mapping $\mathcal{L}$ previously described at the beginning of this section.

3.4 Fitting

The main goal of COMMIT is to solve the inverse formulation of equation 3.4. This means that COMMIT finds the best set $\widetilde{\mathcal{F}} \subseteq \mathcal{F}$ of fibers that best describes the input image I. Our problem appears to be trivial because we only need to invert the matrix A. But for big data sets the matrix A can reach hundreads of gigabytes. So, the size of the matrix A prevents to use classic solutions like pseudo-inverse computation or Cholesky decomposition to estimate the vector x in equation the 3.5. However, in [3.9, 3.10] there were developed iterative least-squares solutions for such large-scale problems. Thus, we have to solve the non-negative least-squares problem

$$\underset{x \geq 0}{argmin} \left\| Ax - y \right\|_2^2 , \tag{3.7}$$

where $\|\cdot\|_2$ is the usual l_2 norm in $\mathbb{R}^{n_s}$ [3.1].

Note that A is not a square matrix. In fact, we usually have a lot more rows than colums. So, the linear system 3.5 does not have a unique solution. But if it is known that the vector x is sparse, we can use a l_1-regularization [3.1]. That changes our problem to

$$\underset{x \geq 0}{argmin} \left\| Ax - y \right\|_2^2 + \left\| x \right\|_1 , \tag{3.8}$$

where $\|\cdot\|_1$ is the usual l_1 norm in $\mathbb{R}^m$. This new formulation promotes sparsity in the solution x.

COMMIT uses the FISTA [3.11] itearative optimization algorithm to solve these problems. FISTA uses the algebraic operations Ax and $A^t y$ during the optimization procedure. If the matrix A is very large, the FISTA algorithm becomes slow. This marks a bottleneck in the COMMIT framework because the matrix A used in COMMIT is very large. That is why we implemented Ax and $A^t y$ by using CUDA because CUDA allows to accelerate the required time to compute these operations.

CHAPTER 4

METHODS AND DEVELOPMENT

Having seen some CUDA tools and the formulation of COMMIT, we can start to talk about the implementation in CUDA. In this chapter, we are going to explain our implementation in CUDA of the algebraic operations Ax and $A^t y$ and the integration of our implementation in the current computational implementation in python of the COMMIT framework.

4.1 Look-up Table with the Precomputed DW-MR Signals

Here we introduce the concept of $look-up\ table$. Look-up tables store all the pre-rotated response functions of all compartments, so we have 3 look-up tables because we have 3 different compartments (IC, EC and ISO). These tables are very important in the implementation because we can avoid to calculate the transformation T_v^f on-the-fly. Instead, we only have to keep an orientation index and go to the look-up table to get the rotated response functions. Of course, it is impossible to store an infinite potential number of orientations for every response function. So, we use a discretization of the half of the sphere shell by using 181 azimuthal and 181 elevation directions. This gives us $n_o = 181^2 = 32,761$ total orientations.

More importantly, look-up tables help to reduce memory usage of the matrix A. When we replace all the response functions of the compartments by an orientation index, we reduce the size of A from $n_v n_s \times (n_f n_a + n_e n_b + n_v n_c)$ to $n_v \times (n_f + n_e + n_v)$.

The intra-cellular look-up table ($lutIC$) should be a three-dimensional float matrix with dimensions $n_a \times n_o \times n_s$. But memory latency is pretty high on GPUs. So, we cannot store $lutIC$ as matrix because we will damage performance by accessing too many pointers. Instead, we have to vectorize (to flatten) this matrix to a $n_a * n_o * n_s$ array and to access by using an offset value. This code ilustrates the change:

```
1  // matrix version ---------------------
2
3  //declaration
4  float lutIC[na][no][ns];
5
6  //accessing element i,j,k
7  lutIC[i][j][k]
8
9  // array version ---------------------
10
11 //declaration
12 float lutIC[na*no*ns];
13
14 //accessing element i,j,k
15 lutIC[i*no*ns + j*ns + k]
```

This technique is the same for $lutEC$ and $lutISO$. In this way, $lutEC$ is changed from a $n_b \times n_o \times n_s$ float matrix to a $n_b * n_o * n_s$ float array and $lutISO$ from a $n_c \times n_s$ float matrix to a $n_c * n_s$ float array.

4.2 Sparse Structure

Even if we use look-up tables, we will not be able to store the matrix A in the GPU memory. For example, our data set used in the next chapter for experiments has $n_v = 64,309$, $n_f = 399,758$, $n_e = 188,151$. Then, our matrix would reach

$$n_v \times (n_f + n_e + n_v) * 4\,B = 64,309 * (399,758 + 188,151 + 64,309) * 4\,B \cong 168\,GB$$

because we represent each entry of the response functions by using 4 bytes single-precision floating-point format. Evidently, there is not a GPU with that amount of memory. Fortunately, our matrix A is a sparse matrix. Moreover, sparsity index [4.5] in the matrix A is usually very high. This property allow us to store the matrix A by using an sparse matrix format.

4.2.1 Block Matrix IC

The matrix A^{IC} has n_a columns for each fiber $f \in \mathcal{F}$ and n_s rows for each voxel $v \in \{0, \ldots, n_v - 1\}$ [3.1]. The n_s rows associated with a voxel v correspond to the restricted signal contribution of all fibers f traversing voxel v. Every restricted contribution of a fiber f traversing voxel v is modeled by transforming the n_a response functions with the operators T_v^f and E_v^f. If a fiber f is not traversing

voxel v, the restricted contribution will be a zero signal. You can see a hypothetical 2D image with its A^{IC} associated matrix in the Figure 4.1.

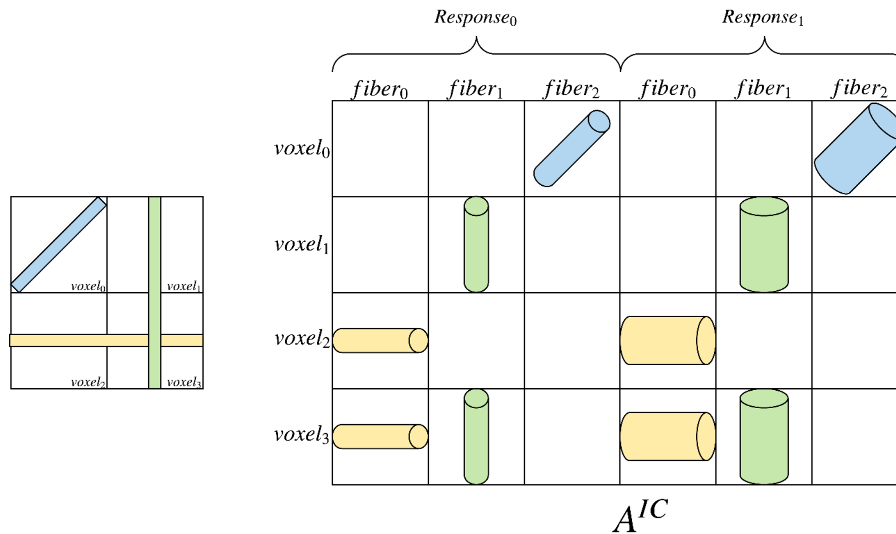

Figure 4.1: Example of a matrix A^{IC}. Blank spaces are zero signals. Cylinder response functions are used.

Even for a very small matrix we have a huge amount of zero values. So, if we use a sparse matrix format, we will reduce the amount of memory needed to store this matrix. In this case, we use a modified version of the coordinate (COO) format. COO stores a tuple of several arrays (row, col and val) [4.1]. The arrays: *row*, *col* and *val* store the row indices, column indices and non-zero values respectively, see Figure 4.2. The total number of non-zero values determinates the size of each array.

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 \end{pmatrix} \longrightarrow \begin{matrix} row & 0 & 1 & 1 & 4 \\ col & 0 & 2 & 3 & 4 \\ val & 2 & 3 & 1 & 9 \end{matrix}$$

Figure 4.2: Example of the COO format for the storage of sparse matrices.

Intra-cellular part of the matrix A depends from the fiber segments passing through the voxels. So, we modified the tuple to ($voxelIC$, $fiberIC$, $contrIC$ and $orienIC$):

1. $voxelIC$: This array stores the voxel index of each fiber segment. The voxel index of one fiber segment determinates the row index in the matrix A.
2. $fiberIC$: This array stores the fiber index of each fiber segment. The fiber index determinates the column index in the matrix A.

3. *contrIC*: This array stores the length of each fiber segment. This length value is used to apply the operator E_v^f introduced in Chapter 3.

4. *orienIC*: This array stores the orientation index of each fiber segment. As we previously saw, this orientation index is used to take the right response functions from the intra-cellular look-up table. This orientation index is used to avoid the computation of the operator T_v^f on-the-fly.

The number of non-zero values in the matrix A is equal to the total number of fiber segments in the tractography. Our data set used for experiments has $47,082,501$ fiber segments and it was designed to be big enough to be considered a good upper bound. We also use 4 bytes unsigned integers for $voxelIC$ and $fiberIC$ arrays, 4 bytes floats for $contrIC$ array and 2 bytes unsigned integers for $orienIC$ array. Therefore, memory usage would be $47,082,501 * 14\,B \cong 754\,MB$, which is affortable on a modern GPU.

4.2.2 Block Matrix EC

Extra-cellular part of the matrix A is independent of the fibers and fiber segments. It entirely depends of the total number of extra-cellular compartments. Then, we only store two arrays ($voxelEC$, $orienEC$):

1. $voxelEC$: This array stores the voxel index of each extra-cellular compartment.

2. $orienEC$: This array stores the orientation index of each extra-celluar compartment.

In this case, the total number of extra-cellular compartments determinates the size of the arrays. For our test data set, we have $n_e = 188,151$. If we use 4 bytes unsigned integers for $voxelEC$ and 2 bytes unsigned integers for $orienEC$, the memory usage would be $188,151 * 6\,B \cong 1.2\,MB$.

4.2.3 Block Matrix ISO

This matrix is even simpler than the previous one. There is only one isotropic compartment per voxel, so we do not need to store voxel indexes. Each compartment is also an isotropic diffusivity model (sphere), so we do not need to store any orientation index. In brief, we do not need to store this matrix in memory. The only thing we need to store is the isotropic look-up table ($lutISO$).

4.3 Preprocessing

At this point, we have stored our matrix A by using the arrays:

- Intra-cellular part (A^{IC}): $voxelIC$, $fiberIC$, $contrIC$, $orienIC$ and $lutIC$.
- Extra-cellular part (A^{EC}): $voxelEC$, $orienEC$ and $lutEC$.
- Isotropic part (A^{ISO}): $lutISO$.

We have all the components to perform the operations Ax and $A^t y$ in the GPU. But in order to take advantage of CUDA and NVIDIA GPUs, we have to preprocess the data for the GPU.

First, we have to sort the intra-cellular and extra-cellular data by using voxel and orientation indexes in order to group fiber segments within the same voxel and with the same orientation. We cannot sort each array independently because the entries of the arrays represent a fiber segment or an extra-cellular compartment. So, we define a *segment* C structure and functions to sort these arrays. The following C code only shows the structure and functions for the intra-cellular part, but the code for extra-cellular part is similar.

```
1  struct segment{
2      uint   voxel;
3      uint   fiber;
4      ushort orien;
5      float  contr;
6  };
7
8  bool vcomp(const segment& a, const segment& b){
9      if (a.voxel != b.voxel) return a.voxel < b.voxel;
10     else                    return a.orien < b.orien;
11 }
12
13 void vsort(uint*   voxelIC, //IC data
14            uint*   fiberIC,
15            ushort* orienIC,
16            float*  contrIC,
17            int num_segments){
18
19     segment* data = (segment*)malloc(num_segments*sizeof(
           segment));
20
21     for(int i = 0; i < num_segments; i++){
22         data[i].voxel = voxelIC[i];
23         data[i].fiber = fiberIC[i];
24         data[i].orien = orienIC[i];
25         data[i].contr = contrIC[i];
26     }
27
28     sort(data, data + num_segments, vcomp);
29
30     for(int i = 0; i < num_segments; i++){
31         voxelIC[i] = data[i].voxel;
32         fiberIC[i] = data[i].fiber;
33         orienIC[i] = data[i].orien;
34         contrIC[i] = data[i].contr;
35     }
36
37     free(data);
38 }
```

We will also calculate a couple of auxiliar arrays for the intra-cellular and extra-cellular compart-

ments. These arrays will make sense later in the next chapter when we introduce the algorithm to perform direct and transposed operations. For now, we will just check what we store in each array:

1. $vsizeIC$: This array has size n_v. The vth entry in this array stores the number of fiber segments in the voxel with id v.

2. $vstepIC$: This array also has size n_v and it stores the all-prefix-sums [4.4] from the array $vsizeIC$. This all-prefix-sums are commonly known as *scan*.

3. $vstepIC$:The length of this array is n_v and it is the exclusive all-prefix-sums [4.4] array calculated from the array $vsizeIC$. The ith entry of this array stores the sum of the previous $i - 1$ elements in the array. For example, if $vsizeIC = [1, 9, 2]$, then we have $vstepIC = [0, 1, 10]$.

The others two arrays ($vsizeEC$ and $vstepEC$) store the same things, but they are calculated from the extra-cellular data. As the isotropic part is explicitly not stored in memory, it does not need these auxiliar arrays. In the code showed down below you can see the preprocessing function we use for IC compartment, but EC function is analogous.

```
1  void preprocessIC(uint* voxelIC, //voxel indexes
2                    uint* vsizeIC, //output
3                    uint* vstepIC, //output
4                    int nv,        //number of voxels
5                    int n){        //number of fiber segments
6      //calculate number of fiber segments per voxel
7      uint count = 0;
8      uint current = voxelIC[0];
9      uint pos = 0;
10     for(int i = 0; i < n; i++){
11         uint aux = voxelIC[i];
12         if(aux != current){
13             vsizeIC[pos] = count;
14             count = 1;
15             pos++;
16             current = aux;
17         }
18         else{
19             count++;
20         }
21     }
22     vsizeIC[pos] = count;
23
24     // calculate all-prefix-sums from vsizeIC
25     vstepIC[0] = 0;
26     for(int i = 1; i < nv; i++)
27         vstepIC[i] = vstepIC[i-1] + vsizeIC[i-1];
28 }
```

4.4 Operation Ax

The intra-cellular, extra-cellular and isotropic parts are independent and they can be calculated separately. Using equation 3.3 we can rewrite equation 3.5 as

$$y = \begin{bmatrix} S_1 \\ \vdots \\ S_{n_v} \end{bmatrix} = Ax = \begin{bmatrix} S_1^{IC} + S_1^{EC} + S_1^{ISO} \\ \vdots \\ S_{n_v}^{IC} + S_{n_v}^{EC} + S_{n_v}^{ISO} \end{bmatrix} = \begin{bmatrix} S_1^{IC} \\ \vdots \\ S_{n_v}^{IC} \end{bmatrix} + \begin{bmatrix} S_1^{EC} \\ \vdots \\ S_{n_v}^{EC} \end{bmatrix} + \begin{bmatrix} S_1^{ISO} \\ \vdots \\ S_{n_v}^{ISO} \end{bmatrix} \in \mathbb{R}^{n_v n_s}.$$

Therefore, we have to calculate $S_v^{IC} \in \mathbb{R}^{n_s}$, $S_v^{EC} \in \mathbb{R}^{n_s}$ and $S_v^{ISO} \in \mathbb{R}^{n_s}$ for every $v \in \{0, \ldots, n_v - 1\}$. As compartments have different structures each other, we will develop a specific algorithm for each compartment.

4.4.1 Algorithm IC

To calculate intra-cellular part, we launch a kernel with n_v CUDA blocks and n_s CUDA threads. Every block is associated with a voxel. This means the CUDA block with identifier v calculates the signal $S_v^{IC} \in \mathbb{R}^{n_s}$. As all CUDA blocks run in parallel, we are calculating all signals S_v^{IC} in parallel. Inside each CUDA block, the n_s entries of the signal S_v^{IC} are computed in parallel by the n_s threads in that CUDA block. Therefore, the thread with identifier k inside the CUDA block with identifier v calculates the value $S_{v,k}^{IC}$ and writes it in y, see Figure 4.3.

$$A^{IC}x = \left[S_0^{IC}, \ldots, S_{n_v-1}^{IC}\right] = \left[\left[S_{0,0}^{IC}, \ldots, S_{0,n_s-1}^{IC}\right], \ldots, \left[S_{n_v-1,0}^{IC}, \ldots, S_{n_v-1,n_s-1}^{IC}\right]\right]$$

Figure 4.3: Distribution of the threads in the algorithm for $A^{IC}x$.

There is not memory collision because all threads are working on different values $S_{v,k}^{IC}$. Remembering equation 3.3, we see that every thread is performing this operation

$$S_{v,k}^{IC} = \sum_{f \in \mathcal{F}} \sum_{j=0}^{n_a-1} x_j^f E_v^f \left(T_v^f \left(R_{j,k}^{IC}\right)\right) \in \mathbb{R}. \tag{4.1}$$

At this point, arrays $vsizeIC$ and $vstepIC$ become important. It is not necessary to traverse across all elements in $\mathcal{F}$ because there are a lot of fibers in $\mathcal{F}$ that do not touch the voxel v. So, these fibers have zero contribution in v, and we can avoid those operations. By instead traversing all fibers in $\mathcal{F}$, we use the elements $vsize\,[v]$ and $vstep\,[v]$. The first one gives us the number of fiber segments voxel v has to process. The second one gives us the position in the intra-cellular data where fiber segments in the voxel v start to appear. Our implementation of the kernel executed by all the threads is showed down below. Note that the kth value of the signal $T_v^f\left(R_j^{IC}\right)$ is not calculated on-the-fly. Instead, we take the value from the IC look-up table.

```
1  __global__ void
2  multiplyIC_direct(uint*   voxelIC,
3                    uint*   fiberIC,
4                    ushort* orienIC,
5                    float*  contrIC,
6                    uint*   vsizeIC,
7                    uint*   vstepIC,
8                    float*  lutIC,
9                    double* x,
10                   double* y){
11     uint v = blockIdx.x;
12     uint k = threadIdx.x;
13     uint start  = vstepIC[v];
14     uint finish = vstepIC[v] + vsizeIC[v];
15
16     double Svk = 0.0;
17     for(int f = start; f < finish; f++){ //loop first  sum
18     for(int j = 0; j < gpu_na; j++){     //loop second sum
19         //look-up table offset
20         int offset = j*gpu_no*gpu_ns + orienIC[f]*gpu_ns + k;
21
22         Svk += (double)(lutIC[offset])*  //T(R)
23                (double)(contrIC[f])*     //E(T(R))
24                x[fiberIC[f] + j*gpu_nf]; //x*E(T(R))
25     }}
26
27     //write Svk
28     y[voxelIC[start]*gpu_ns + k] += Svk;
29 }
```

The values *gpu_na*, *gpu_no*, *gpu_ns* are the values n_a, n_o and n_s stored in constant GPU memory. We have to set these values before to launch this kernel. These values do not have to be passed by argument to the kernel because variables in constant memory are global.

4.4.2 Algorithm EC

The extra-cellular part is analogous to the intra-cellular part. Moreover, this part is simpler than the intra-cellular part. In this algorithm, CUDA blocks and threads are assigned in the same way as the previous algorithm. So, each thread is computing the following operation taken from the equation 3.3

$$S_{v,k}^{EC} = \sum_{e \in \mathcal{O}} \sum_{j=0}^{n_b - 1} x_j^e T_v^e \left(R_{j,k}^{EC} \right) \in \mathbb{R}.$$

In this case, we do not have to apply operator E_v^f. We also have to calculate the pointer xEC which points to the extra-cellular part of the vector x. Implementation of this part is shown below.

```
1  __global__ void
2  multiplyEC_direct(uint*   voxelEC,
3                    ushort* orienEC,
4                    uint*   vsizeEC,
5                    uint*   vstepEC,
6                    float*  lutEC,
7                    double* x,
8                    double* y){
9      uint v = blockIdx.x;
10     uint k = threadIdx.x;
11
12     uint start  = vstepEC[v];
13     uint finish = vstepEC[v] + vsizeEC[v];
14
15     //pointer to the EC part of array x
16     double* xEC = x + gpu_nf*gpu_na + start;
17
18     double Svk = 0.0;
19     for(int e = start; e < finish; e++){ //loop first  sum
20     for(int j = 0; j < gpu_nb; j++){     //loop second sum
21         //look-up table offset
22         int offset = j*gpu_no*gpu_ns + orienEC[e]*gpu_ns + k;
23
24         Svk += (double)(lutEC[offset])* //T(R)
25                xEC[e + j*gpu_ne];       //x*T(R)
26     }}
27
28     //write Svk
29     y[voxelEC[start]*gpu_ns + k] += Svk;
30 }
```

4.4.3 Algorithm ISO

Assignment of CUDA blocks and threads are exactly the same as the previous one. So, each thread is computing this operation

$$S^{ISO}_{v,k} = \sum_{j=0}^{n_c-1} x^v_j R^{ISO}_{j,k} \in \mathbb{R} \tag{4.2}$$

which is also part of the equation 3.3. Isotropic part is the simplest one. In this case, we do not even have to apply any operator because isotropic response functions are rotationally invariant and there is only one isotropic contribution per voxel. We also calculate the pointer $xISO$ which point to the isotropic part of the vector x. The code is showed below:

```
1  __global__ void
2  multiplyISO_direct(float*  lutISO,
3                     double* x,
4                     double* y){
5      uint v = blockIdx.x;
6      uint k = threadIdx.x;
7
8      //pointer to the ISO part of the vector x
9      double* xISO = x + gpu_nf*gpu_na + gpu_ne*gpu_gpu_nb + k;
10
11     double Svk = 0.0;
12     for(int j = 0; j < gpu_nc; j++){
13         //look-up table offset
14         int offset = j*gpu_ns + k;
15
16         Svk += (double)(lut[offset])* //R
17                xISO[j*gpu_nv];        //x*R
18     }
19
20     //write Svk
21     y[v*gpu_ns + k] += Svk;
22 }
```

4.5 Operation $A^t y$

The structure of the transposed version of the operator A is irrelevant for COMMIT model. For this reason, we will not spend time explaining the structure of A^t. But to be able to understand

the implementation of this operation, it is important to talk about some formulations.

We have seen that $A \in \mathbb{R}^{n_v n_s \times m}$ is a block matrix composed of column matrices A^{IC}, A^{EC} and A^{ISO}. In this case, $A^t \in \mathbb{R}^{m \times n_v n_s}$ is also a block matrix, but it is composed of row matrices $A^{tIC} \in \mathbb{R}^{n_f n_a \times n_v n_s}$, $A^{tEC} \in \mathbb{R}^{n_e n_b \times n_v n_s}$ and $A^{tISO} \in \mathbb{R}^{n_v n_c \times n_v n_s}$. Thus, we have

$$A^t = \begin{bmatrix} A^{tIC} \\ - \\ A^{tEC} \\ - \\ A^{tISO} \end{bmatrix}.$$

As the operation Ax, we will present three different algorithms for each IC, EC and ISO compartments.

4.5.1 Algorithm IC

Having seen the structure and implementation of $A^{IC}x$, we know that we had to compute all the signals S_v^{IC} for every $v \in \{0, \ldots, n_v - 1\}$. For this case, after some large algebraic manipulations using the definition of the transpose of a matrix, we get that

$$A^{tIC} y = \begin{bmatrix} Z_0^{IC} \\ \vdots \\ Z_{n_f-1}^{IC} \end{bmatrix} \in \mathbb{R}^{n_f n_a}, \tag{4.3}$$

where we have to compute all the signals $Z_f^{IC} \in \mathbb{R}^{n_a}$ for every fiber $f \in \mathcal{F}$ to fill the following vectors

$$Z_f^{IC} = \begin{bmatrix} Z_{f,0,0}^{IC} + Z_{f,0,1}^{IC} + \cdots + Z_{f,0,n_s-1}^{IC} \\ Z_{f,1,0}^{IC} + Z_{f,1,1}^{IC} + \cdots + Z_{f,1,n_s-1}^{IC} \\ \vdots \\ Z_{f,n_a-2,0}^{IC} + Z_{f,n_a-2,1}^{IC} + \cdots + Z_{f,n_a-2,n_s-1}^{IC} \\ Z_{f,n_a-1,0}^{IC} + Z_{f,n_a-1,1}^{IC} + \cdots + Z_{f,n_a-1,n_s-1}^{IC} \end{bmatrix} \in \mathbb{R}^{n_a}, \tag{4.4}$$

with

$$Z_{f,j,k}^{IC} = \sum_{v=0}^{n_v-1} y_j^v E_v^f \left(T_v^f \left(R_{j,k}^{IC} \right) \right) \in \mathbb{R}. \tag{4.5}$$

To launch the kernel of this part, we use n_f CUDA blocks and n_s CUDA threads. The block with identifier f computes the signal Z_f^{IC}. Inside each CUDA block, the n_s values $Z_{f,j,k}^{IC}$ of the jth row are computed by the n_s threads in parallel, see Figure 4.4.

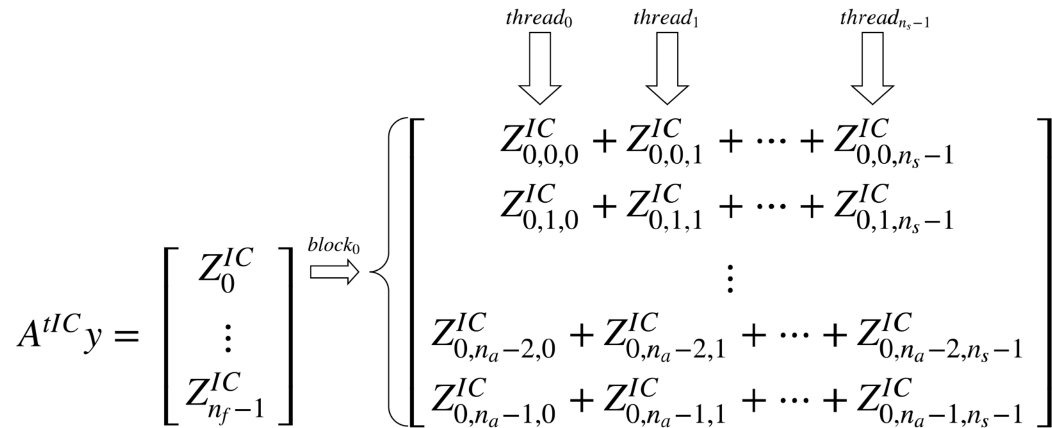

Figure 4.4: Distribution of CUDA blocks and CUDA threads in the computation of A^{tIC}.

Since threads 0 through $n_s - 1$ compute, for a given $j \in \{0, \ldots, n_a - 1\}$ and $f \in \mathcal{F}$, signals $Z_{f,j,0}^{IC}$ through Z_{f,j,n_s-1}^{IC} respectively, we need to use shared memory to sum all these signals before writting the final value. There are two ways to sum these values:

1. To use the thread with identifier 0. This thread loads all the values from shared memory, sequentially sums all of them and writes the result. But this approach is very inefficient because the $n_s - 1$ remaining threads have to wait doing nothing until this thread finishes.

2. To use the parallel reduction [2.6, 4.4]. In short, this approach takes advantage of the multiple threads to compute the result. Every thread is assigned to two elements in the array (if it is possible). Then, each thread loads those two elements and writes back to shared memory. This process is repeated until we have a single value which is the final result. Figure 4.5 shows a graphical example of this process.

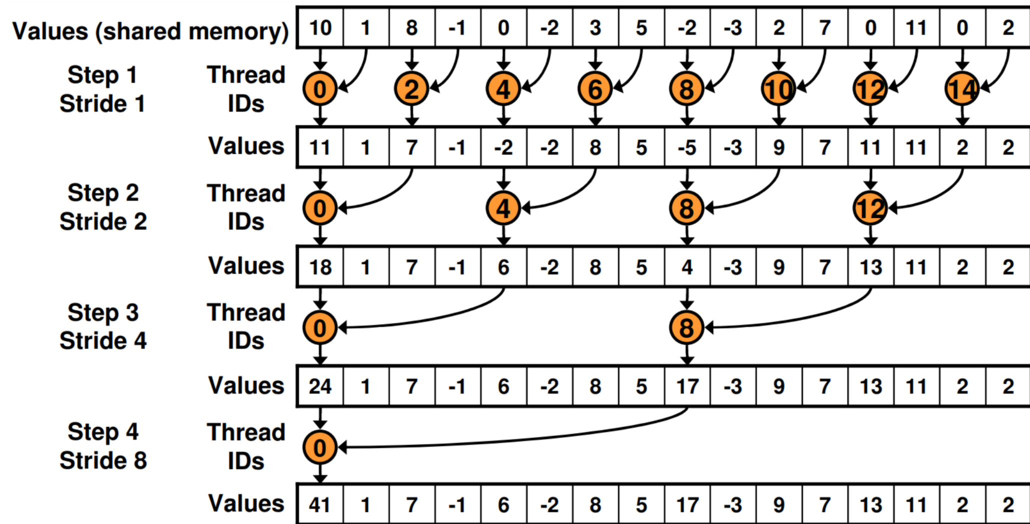

Figure 4.5: Steps of a parallel reduction. Each orange circle indicates an operation performed by a thread. This image is property of NVIDIA Corporation.

Before seeing the code for this part, it is important to say that the arrays $voxelIC$, $fiberIC$, $orienIC$ and $contrIC$ composing the sparse structure of A have to be re-sorted by using the fiber index. This helps to improve performance in the multiplication. So, we have a new tuple ($voxeltIC$, $fibertIC$, $orientIC$ and $contrtIC$) forming the sparse structure of A^{tIC}. The code to sort these arrays is the same we saw in the last section. We only have to change the comparison function for the following one:

```
1 bool fcomp(const segment& a, const segment& b){
2     if (a.fiber != b.fiber) return a.fiber < b.fiber;
3     else                    return a.orien < b.orien;
4 }
```

Moreover, the equation 4.5 can be optimized. This equation considers that every fiber f is traversing all the n_v voxels. For real data this is not true. In fact, fibers just traverse a few amount of voxels. If a fiber f does not intersect a voxel v, this voxel contributes with a zero-signal. So, like the direct multiplication we can precalculate two extra auxiliar arrays ($fsizetIC$ and $fsteptIC$).

1. $fsizetIC$: The length of this array is n_f. The fth entry in this array stores the number of voxels intersected by the fiber f.

2. $fsteptIC$: The length of this array is n_f and it is the exclusive all-prefix-sums [4.4] array calculated from the array $fsizetIC$. The ith entry of this array stores the sum of the previous $i-1$ elements in the array. For example, if $fsizetIC = [1, 9, 2]$, then we have $fsteptIC = [0, 1, 10]$.

Subsequently, these arrays are analogous to the arrays $vsizeIC$ and $vstepIC$. Moreover, the code to calculate them is almost the same because you only have to swap $voxelIC$ for $fiberIC$ and n_v for n_f in the function $preprocessIC(\ldots)$ showed at the beginning of this chapter. Finally, below is the code for the operation $A^{tIC}y$:

```
1  __global__ void
2  multiplyIC_transp(uint*   voxeltIC,
3                    uint*   fibertIC,
4                    ushort* orientIC,
5                    float*  contrtIC,
6                    uint*   fsizetIC,
7                    uint*   fsteptIC,
8                    float*  lutIC,
9                    double* x,
10                   double* y){
11 __shared__ double shsum[512];
12
13 uint f = blockIdx.x;
14 uint k = threadIdx.x;
15
16 uint start  = fsteptIC[f];
17 uint finish = fsteptIC[f] + fsizetIC[f];
18
19 for(int j = 0; j < gpu_na; j++){
20     double Zfjk = 0.0;
21     for(int v = start; v < finish; v++){
22         //look-up table offset
23         int offset = j*gpu_no*gpu_ns + orientIC[v]*gpu_ns + k;
24         Zfjk += (double)(lutIC[offset])*    //T(R)
25                 (double)(contrtIC[v])*      //E(T(R))
26                 y[voxeltIC[v]*gpu_ns + k]; //y*E(T(R))
27     }
28     //each thread write its Zfjk value
29     shsum[k] = Zfjk; __syncthreads();
30
31     if(k<256) shsum[k] += shsum[k+256]; __syncthreads();
32     if(k<128) shsum[k] += shsum[k+128]; __syncthreads();
33     if(k< 64) shsum[k] += shsum[k+ 64]; __syncthreads();
34     if(k< 32) shsum[k] += shsum[k+ 32]; __syncthreads();
35     if(k< 16) shsum[k] += shsum[k+ 16]; __syncthreads();
36     if(k<  8) shsum[k] += shsum[k+  8]; __syncthreads();
37     if(k<  4) shsum[k] += shsum[k+  4]; __syncthreads();
38     if(k<  2) shsum[k] += shsum[k+  2]; __syncthreads();
39
40     if(k==0) x[j*gpu_nf + fibertIC[start]] = shsum[0]+shsum[1];
41 }}
```

4.5.2 Algorithm EC

In contrast to the fibers (intra-cellular compartments), each extra-cellular compartment is only associated with a single voxel. In this manner, we do not need to re-sort the sparse structure of A^{EC} (arrays $voxelEC$ and $orienEC$). Thus, we can compute $A^{tEC}y$ by using a very similar kernel to the direct multiplication. Due to we are working with the transposed version, we have to change the order in the indexes as well as we use the parallel reduction. The code to compute $A^{tEC}y$ is in the box below:

```
1  __global__ void
2  multiplyEC_transp(uint*   voxelEC,
3                    ushort* orienEC,
4                    uint*   vsizeEC,
5                    uint*   vstepEC,
6                    float*  lutEC,
7                    double* x,
8                    double* y){
9      __shared__ double shsum[512];
10
11     uint v = blockIdx.x;
12     uint k = threadIdx.x;
13
14     uint start  = vstep[v];
15     uint finish = vstep[v] + vsize[v];
16
17     double* xEC = x + gpu_nf*gpu_na;
18
19     for(int j = 0; j < gpu_nb; j++){
20     for(int e = start; e < finish; e++){
21         //look-up table offset
22         int offset = j*gpu_no*gpu_ns + orienEC[e]*gpu_ns + k;
23
24         //compute Svk and write it to shared memory
25         shsum[k] = (double)(lutEC[offset])*      //T(R)
26                    y[voxelEC[start]*gpu_ns + k]; //y*T(R)
27         __syncthreads();
28
29         if(k<256) shsum[k] += shsum[k+256]; __syncthreads();
30         if(k<128) shsum[k] += shsum[k+128]; __syncthreads();
31         if(k< 64) shsum[k] += shsum[k+ 64]; __syncthreads();
32         if(k< 32) shsum[k] += shsum[k+ 32]; __syncthreads();
33         if(k< 16) shsum[k] += shsum[k+ 16]; __syncthreads();
34         if(k<  8) shsum[k] += shsum[k+  8]; __syncthreads();
35         if(k<  4) shsum[k] += shsum[k+  4]; __syncthreads();
36         if(k<  2) shsum[k] += shsum[k+  2]; __syncthreads();
37
38         if(k==0) xEC[j*gpu_ne + e] += shsum[0]+shsum[1];
39     }}
40 }
```

4.5.3 Algorithm ISO

Undoubtedly, the isotropic part is even simpler because there is only one isotropic compartment per voxel. Indeed, we have to change the indexes because of the transposed version and we use the parallel reduction as in the previous algorithms.

```
1  __global__ void
2  multiplyISO_transp(float* lut,
3                     double* x,
4                     double* y){
5      __shared__ double shsum[512];
6
7      uint v = blockIdx.x;
8      uint k = threadIdx.x;
9
10     //pointer to the ISO part of the vector x
11     double* xISO = x + gpu_nf*gpu_na + gpu_ne*gpu_nb;
12
13     for(int j = 0; j < gpu_nc; j++){
14         //look-up table offset
15         int offset = j*gpu_ns + k;
16
17         //compute Svk and write it shared memory
18         shsum[k] = (double)(lut[offset])* //R
19                    y[bid*gpu_ns + k];     //y*R
20         __syncthreads();
21
22         if(k<256) shsum[k] += shsum[k+256]; __syncthreads();
23         if(k<128) shsum[k] += shsum[k+128]; __syncthreads();
24         if(k< 64) shsum[k] += shsum[k+ 64]; __syncthreads();
25         if(k< 32) shsum[k] += shsum[k+ 32]; __syncthreads();
26         if(k< 16) shsum[k] += shsum[k+ 16]; __syncthreads();
27         if(k<  8) shsum[k] += shsum[k+  8]; __syncthreads();
28         if(k<  4) shsum[k] += shsum[k+  4]; __syncthreads();
29         if(k<  2) shsum[k] += shsum[k+  2]; __syncthreads();
30
31         if(k==0) xISO[j*gpu_nv + v] += shsum[0]+shsum[1];
32     }
33 }
```

4.6 Improving the Speed of the Ax operation

The algorithm introduced to compute $A^{IC}x$ (the IC part of the operation Ax) has a bottleneck. In that algorithm, each block computes the signal contribution of every fiber segment inside its assigned voxel. But the number of fiber segments in each voxel is different. Moreover, the number of fiber segments per voxel is very unbalanced. For example, there are voxels with less than 10 fiber segments compared with others having more than $12,000$ fiber segments. So, we will have CUDA blocks finishing quickly and waiting for the others. This is a bottleneck because the kernel only finishes when all CUDA blocks finish their operations. Thus, the kernel execution time for $A^{IC}x$ is determined by the slower CUDA block which is the CUDA block with the highest number of fiber segments.

In this section, we will introduce a new algorithm to calculate $A^{IC}x$ which attempts to reduce the number of operations performed by every CUDA block. The kernel presented in section 4.4.1 is launched with n_v CUDA blocks and n_s CUDA threads. Now, this new kernel will be launched with n_v CUDA blocks and $2n_s$ CUDA threads. We divide the fiber segments in each voxel into two groups. Then, in every CUDA block with identifier v the first n_s CUDA threads compute $S^{IC}_{v,k}$ for the first half of the fiber segments in the voxel v and the second n_s CUDA threads compute $S^{IC}_{v,k}$ for the second half. We need to use shared memory to store these two preliminar values before to write them in the resulting vector y. The code box below shows the code for this new approach.

```
1  __global__ void
2  multiplyIC_direct(uint*   voxelIC,
3                    uint*   fiberIC,
4                    ushort* orienIC,
5                    float*  contrIC,
6                    uint*   vsizeIC,
7                    uint*   vstepIC,
8                    float*  lutIC,
9                    double* x,
10                   double* y){
11     __shared__ double shmem[1024];
12
13     uint v = blockIdx.x;
14     uint k = threadIdx.x;
15     uint gid = threadIdx.x / gpu_ns;      //group id (0 or 1)
16     uint sid = threadIdx.x - gpu_ns*gid; //sample id
17     uint start  = vstepIC[v] + (vsizeIC[v]/2)*gid;
18     uint finish = start + vsizeIC[v]/2 + (vsizeIC[v]%2)*gid;
19
20     double Svk = 0.0;
21     for(int f = start; f < finish; f++){ //loop first  sum
22     for(int j = 0; j < gpu_na; j++){      //loop second sum
23         //look-up table offset
24         int offset = j*gpu_no*gpu_ns + orienIC[f]*gpu_ns + sid;
25
26         Svk += (double)(lutIC[offset])* // T(R)
27                (double)(contrIC[f])*    // E(T(R))
28                x[fiber[f] + j*gpu_nf];  // x*E(T(R))
29     }}
30
31     //every thread load Svk to shared memory
32     shmem[k] = Svk;
33     __syncthreads();
34
35     //only the first group write the final value
36     if(gid == 0)
37         y[voxel[start]*gpu_ns + sid] = Svk + shmem[k+gpu_ns];
38 }
```

This new algorithm decreases the required time to calculate $A^{IC}x$ because the amount of fiber segments per CUDA blocks are reduced in half. In particular, the CUDA block with the highest

number of fiber segments which determines the required time to calculate $A^{IC}x$.

We could extend the previous idea to support more than two groups of CUDA threads. However, we observe that the use of more than two groups will not be supported by the current available NVIDIA hardware. For example, the largest data set available right now has $n_s = 512$. So, if we use more than two groups of CUDA threads, we will overflow the maximum number of CUDA threads permitted in a CUDA block which is 1024.

4.7 Integration with COMMIT

We could implement the operations Ax and $A^t y$ in some python cuda module like PyCUDA. But in order to gain as much performance as possible, we will keep our implementation written in CUDA C++ and we will use the ctypes python module to launch CUDA C++ code from python. ctypes allows to call foreign functions in DLLs or shared libraries and it provides C compatible data types [4.3]. In ctype there is not CUDA specific data types. But we can use the fact that CUDA and C pointers are very similar as we saw in the Chapter 2 of this document. So, we developed a custom CUDA module by using ctypes. We called this module cudac. This module includes functions to allocate and track CUDA memory pointers from python. The kernels for multiplication and preprocessing functions are also packed in this cudac module.

We are not going to discuss the implementation of COMMIT in CPU. But it is important to show how COMMIT works in order to talk about which parts of COMMIT we modified. Figure 4.6 shows a common diagram of a model fitting using COMMIT. It does not involve all the necessary steps to run the fitting, but it provides an idea of the fitting process performed by COMMIT framework.

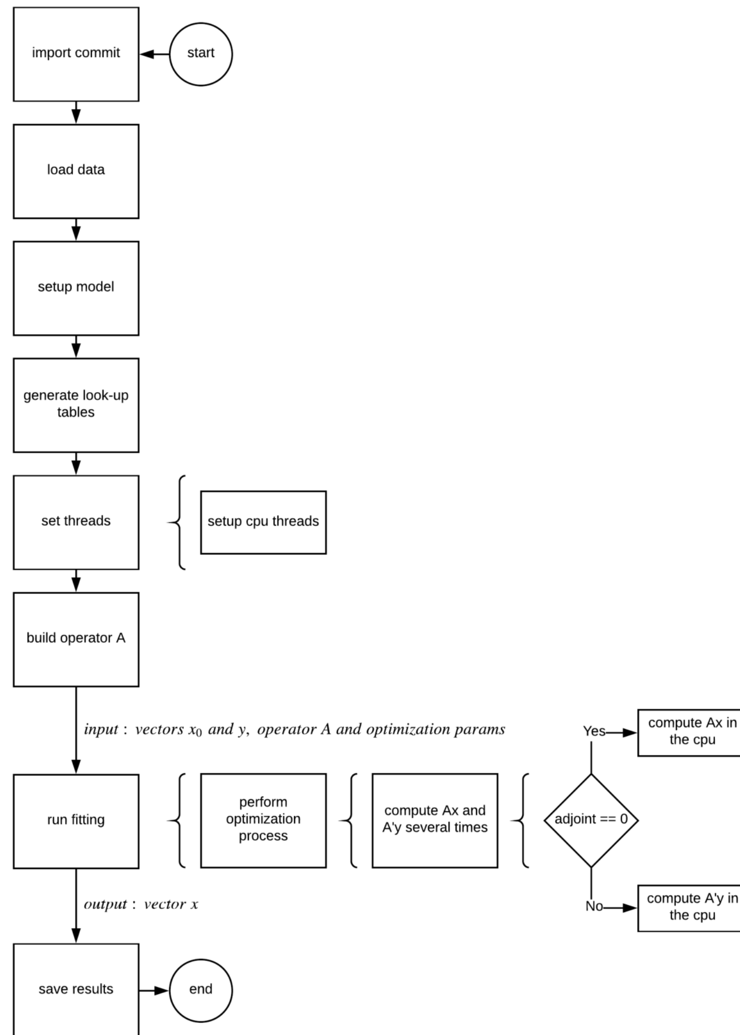

Figure 4.6: Diagram of a simple fitting process using COMMIT framework. Curly brackets show some of the process inside blocks.

The first change was to add a "cuda flag". If this flag is set to 1, COMMIT will use our implementation in CUDA to compute the algebraic operations Ax and $A^t y$. On the other hand, if this flag is equal to 0, COMMIT will normally continue with the CPU version. The use of this cuda flag keeps compatibility with previous COMMIT python scripts.

Another important change was made in the block "set threads". In this block, we initialize and copy all the necessary variables to the constant GPU memory, see the code box below. Please note that the function is wrapped by *extern "C"* which is required to execute this function from Python by using ctypes.

```
1  extern "C" {
2  void set_globals(int num_segments,
3                   int num_voxels,
4                   int num_fibers,
5                   int num_excomp,
6                   int num_orient,
7                   int num_samples,
8                   int num_resfunic,
9                   int num_resfunec,
10                  int num_resfunis,
11                  int num_rows,
12                  int num_cols){
13
14     //initialize variables in CPU
15     n  = num_segments; //number of fiber segments
16     nv = num_voxels;   //number of voxels
17     nf = num_fibers;   //number of fibers
18     ne = num_excomp;   //number of EC compartments
19     no = num_orient;   //number of orientations
20     ns = num_samples;  //number of direction samples
21     na = num_resfunic; //number of IC  response func.
22     nb = num_resfunec; //number of EC  response func.
23     nc = num_resfunis; //number of ISO response func.
24     nrows = num_rows;  //number of rows of the operator A
25     ncols = num_cols;  //number of cols of the operator A
26
27     //initialize variables in the GPU constant memory
28     cudaMemcpyToSymbol(gpu_n,  &n,  sizeof(int));
29     cudaMemcpyToSymbol(gpu_nv, &nv, sizeof(int));
30     cudaMemcpyToSymbol(gpu_nf, &nf, sizeof(int));
31     cudaMemcpyToSymbol(gpu_ne, &ne, sizeof(int));
32     cudaMemcpyToSymbol(gpu_no, &no, sizeof(int));
33     cudaMemcpyToSymbol(gpu_ns, &ns, sizeof(int));
34     cudaMemcpyToSymbol(gpu_na, &na, sizeof(int));
35     cudaMemcpyToSymbol(gpu_nb, &nb, sizeof(int));
36     cudaMemcpyToSymbol(gpu_nc, &nc, sizeof(int));
37 }}
```

In the same block we preprocess the data of the blocks matrices A^{IC}, A^{EC} and A^{ISO} and we create all the auxiliar arrays using the function that we introduced earlier in this chapter. Then, we have to transfer this data to the GPU. Keeping pointers to the location of this data in the GPU is

essential to get a good performance because we prevent to copy all this data in every computation of the algebraic operations Ax and $A^t y$. Our cudac module has an object called DeviceMemory. This object stores, tracks and keeps together all the pointers to the sparse structures, look-up tables and auxiliar vectors. After this modifications, the diagram of the new process is showed in Figure 4.7.

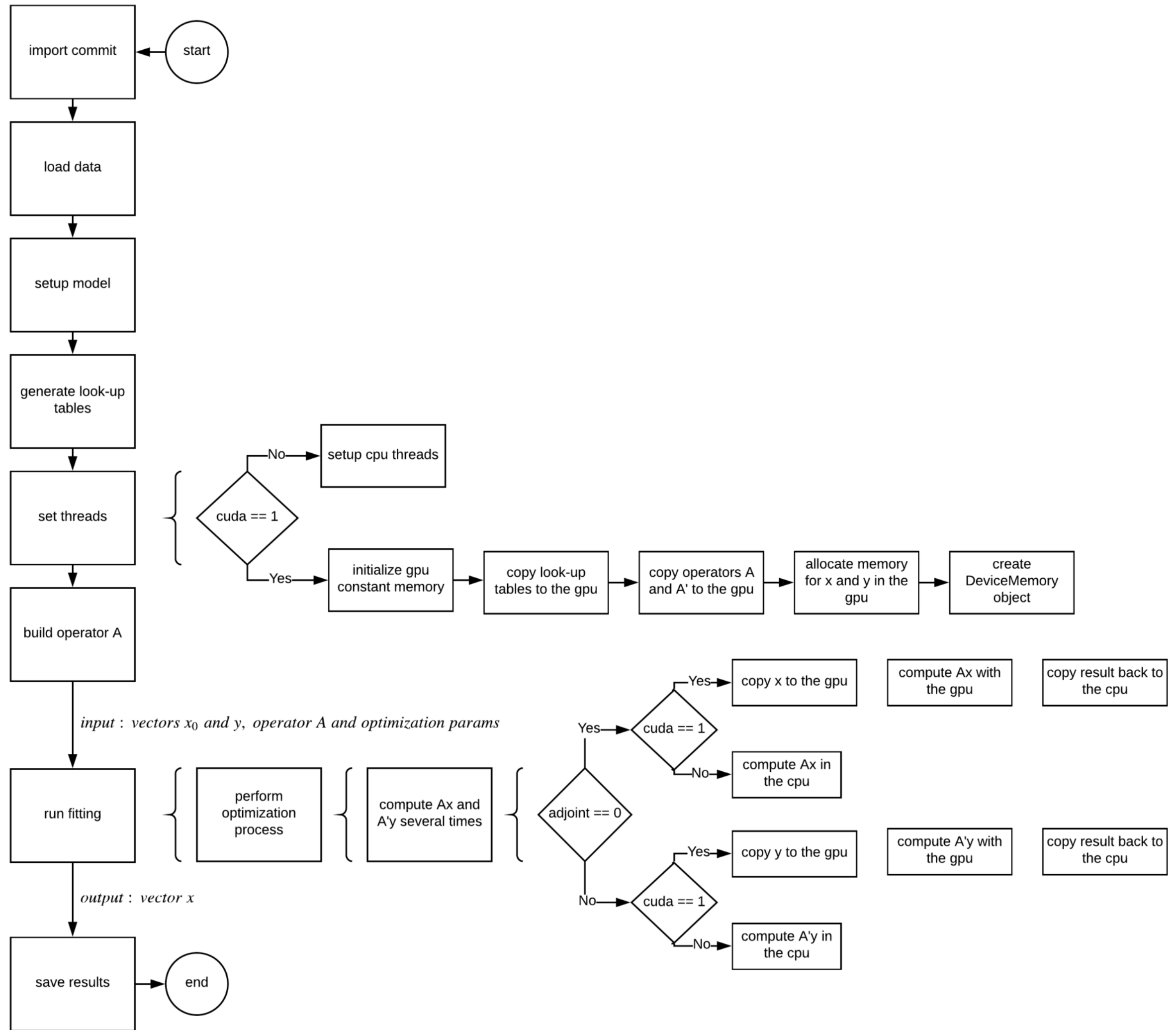

Figure 4.7: Block diagram showing a fitting using the modified COMMIT. Curly brackets show some of the process inside each block.

The object DeviceMemory also stores two aditional pointers. These pointers point to two functions written in C which launch the operations Ax and $A^t y$ in the GPU. Evidently, both functions use the kernels introduced in the previous sections and the code for these functions is showed in the box below:

```
1  extern "C" {
2  void operationAX(
3      uint*   gpu_voxelIC, //IC data
4      uint*   gpu_fiberIC,
5      ushort* gpu_orienIC,
6      float*  gpu_contrIC,
7      uint*   gpu_vsizeIC,
8      uint*   gpu_vstepIC,
9      float*  gpu_lutIC,
10     uint*   gpu_voxelEC, //EC data
11     ushort* gpu_orienEC,
12     uint*   gpu_vsizeEC,
13     uint*   gpu_vstepEC,
14     float*  gpu_lutEC,
15     float*  gpu_lutISO,  //ISO data
16     double* gpu_x,  //vector x in gpu
17     double* gpu_y,  //vector y in gpu
18     double* x,      //vector x in cpu
19     double* y){     //vector y in cpu
20
21     //copy vector x to gpu
22     cudaMemcpy(gpu_x, x, ncols*sizeof(double),
           cudaMemcpyHostToDevice);
23
24     //perform y += A^IC*x in the gpu
25     multiplyIC_direct<<<nv, ns>>>(gpu_voxelIC, gpu_fiberIC,
           gpu_orienIC, gpu_contrIC, gpu_vsizeIC, gpu_vstepIC,
           gpu_lutIC, gpu_x, gpu_y);
26
27     //perform y += A^EC*x in the gpu
28     multiplyEC_direct<<<nv, ns>>>(gpu_voxelEC, gpu_orienEC,
           gpu_vsizeEC, gpu_vstepEC, gpu_lutEC, gpu_x, gpu_y);
29
30     //perform y += A^ISO*x in the gpu
31     multiplyISO_direct<<<nv, ns>>>(gpu_lutISO, gpu_x, gpu_y);
32
33     //copy back result to cpu
34     cudaMemcpy(y, gpu_y, nrows*sizeof(double),
           cudaMemcpyDeviceToHost);
35 }}
```

```
1  extern "C" {
2  void operationATX(
3       uint*   gpu_voxeltIC, //IC data
4       uint*   gpu_fibertIC,
5       ushort* gpu_orientIC,
6       float*  gpu_contrtIC,
7       uint*   gpu_fsizetIC,
8       uint*   gpu_fsteptIC,
9       float*  gpu_lutIC,
10      uint*   gpu_voxelEC, //EC data
11      ushort* gpu_orienEC,
12      uint*   gpu_vsizeEC,
13      uint*   gpu_vstepEC,
14      float*  gpu_lutEC,
15      float*  gpu_lutISO, //ISO data
16      double* gpu_x,  //vector x in gpu
17      double* gpu_y,  //vector y in gpu
18      double* x,      //vector x in cpu
19      double* y){     //vector y in cpu
20
21      //copy vector y to gpu
22      cudaMemcpy(gpu_y, y, nrows*sizeof(double),
            cudaMemcpyHostToDevice);
23
24      //perform x += A^tIC*y in the gpu
25      IC_multiply_transp<<<nf, ns>>>(gpu_voxeltIC, gpu_fibertIC,
            gpu_orientIC, gpu_contrtIC, gpu_fsizetIC, gpu_fsteptIC,
             gpu_lutIC, gpu_x, gpu_y);
26
27      //perform x += A^tEC*y in the gpu
28      EC_multiply_transp<<<nv, ns>>>(gpu_voxelEC, gpu_orienEC,
            gpu_vsizeEC, gpu_vstepEC, gpu_lutEC, gpu_x, gpu_y);
29
30      //perform x += A^tISO*y in the gpu
31      ISO_multiply_transp<<<nv, ns>>>(gpu_lutISO, gpu_x, gpu_y);
32
33      //copy back result to cpu
34      cudaMemcpy(x, gpu_x, ncols*sizeof(double),
            cudaMemcpyDeviceToHost);
35 }}
```

CHAPTER 5

EXPERIMENTS AND RESULTS

In this chapter we will describe the experiments that were run to compare the performance between the CPU version and our GPU implementation in CUDA. The experiments are divided in two parts: The performance of the operations Ax and $A^t y$ and the performance during the fitting process. For the operations Ax and $A^t y$ we used random vectors x and y. For the model fitting we use real human brain data provided by the Dr. Alessandro Daducci and acquired at his laboratory in the University of Verona. The workstation used for the experiments is equiped with an AMD Threadripper 1950x with 32GB DDR4 memory along with a NVIDIA Quadro P6000 with 24GB GDDR5X memory.

5.1 Operation Ax

As explained in the Chapter 4, the integration of our code into the COMMIT framework allows for compatibility with previous COMMIT python scripts. Thus, to test operation Ax we execute the same script for both CPU and GPU versions. The only difference is the argument N in the member $set_threads(N)$ located in line 22 of the code box below. This N value indicates how many CPU threads COMMIT will use to compute the operation Ax. If $0 < N < 255$ COMMIT will use (if it is possible) N threads to compute operation Ax. If $set_threads()$ receives a null argument, COMMIT will use all the available CPU threads to compute Ax. But, if we set $N = 0$, it means that COMMIT will use 0 CPU threads, i.e. it will only use our CUDA implementation of the operation Ax. When we set $N = 0$, COMMIT set the "cuda flag" to 1, see Figure 4.7. In addition, to test performance of the operation Ax we use random vectors x generated with the random python module and each vector has float values between 0 and 1.

```
1  import commit
2  import numpy as np
3
4  commit.core.setup()
5  mit = commit.Evaluation('.', '.')
6  mit.set_config('doMergeB0', False)
7  mit.set_config('doNormalizeKernels', True)
8  mit.set_config('doNormalizeSignal', True)
9  mit.load_data('empty.nii.gz', 'dwi.scheme')
10
11 # set model
12 mit.set_model('CylinderZeppelinBall')
13 mit.model.set(1.7E-3, np.linspace(1,5.0,9)*1E-6, np.arange
       (0.4,0.8,0.1), np.array([3.0E-3, 7.0E-3]))
14
15 # generate and load look-up tables
16 mit.generate_kernels(regenerate=False)
17 mit.load_kernels()
18
19 mit.load_dictionary('CommitOutput')
20
21 # setting 0 threads means we are using gpu
22 mit.set_threads(0)
23
24 # build operator A
25 mit.build_operator()
26
27 # number of tests
28 num_tests = 50
29 for i in xrange(0, num_tests):
30     x = np.random.rand(A.shape[1]).astype('float64')
31     start  = time.time()
32     y = A.dot(x)
33     finish = time.time()
34     print 'Operation␣Ax␣time:␣', (finish - start)
```

The AMD Threadripper 1950x has 32 threads running at 3.4GHz. This allows to compare the performance of our CUDA version versus the CPU version by using 1, 2, 4, 8, 16 and 32 CPU threads, see Figure 5.1. In the Table 5.1 the average times of 50 experiments are showed.

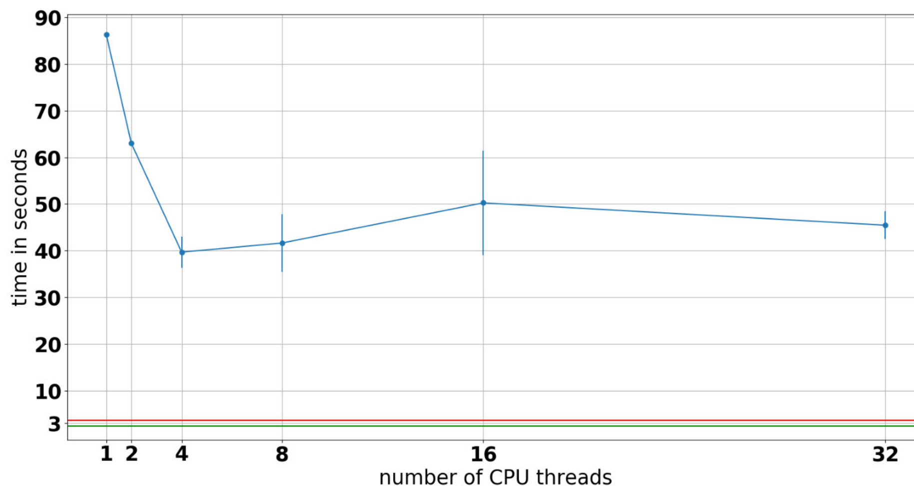

Figure 5.1: The computational time required by the operation Ax using 1, 2, 4, 8, 16 and 32 CPU threads. Horizontal axis represents the number of CPU threads and vertical axis represents the average time in seconds of the 50 experiments. The blue vertical lines represent the variance in the 50 experiments. The red line shows the time of the GPU version. Green line shows the GPU version with the modified IC algorithm introduced in the Section 4.6.

N. Threads	1	2	4	8	16	32	CUDA	Improved CUDA
Time (seconds)	86.3	62.97	39.67	41.63	50.23	45.44	3.62	2.35

Table 5.1: Average times of the experiments.

In order to corrobate that the parallel computed operation Ax is correct, we set the same seed for the CPU and GPU, and we compared the output vector y obtained with the GPU version versus the output vector y obtained with the CPU version. Figure 5.2 shows the relative error $\|y_{gpu} - y_{cpu}\| \,/\, \|y_{gpu}\|$. For the operation Ax, the value $\|y_{gpu} - y_{cpu}\| \,/\, \|y_{gpu}\|$ is virutally zero which means that our implementation of the operation Ax in CUDA gives accurate results.

	Relative error: $\frac{\|y_{cpu}-y_{gpu}\|}{\|y_{gpu}\|}$
1 thread	$2.06 * 10^{-15}$
2 threads	$2.06 * 10^{-15}$
4 threads	$2.06 * 10^{-15}$
8 threads	$2.06 * 10^{-15}$
16 threads	$2.06 * 10^{-15}$
32 threads	$2.06 * 10^{-15}$

Table 5.2: Relative error between vectors y_{gpu} and y_{cpu}. The vector y_{gpu} represents the output vector obtained with the GPU version of the operation Ax and y_{cpu} represents the output vector obtained with CPU version of Ax.

As we mentioned in the Section 3.4, during the fitting procedure COMMIT promotes the sparsity in the vector x. This aspect was considered in the CPU implementation of the operation Ax. So,

the CPU version of COMMIT checks for zero values in the vector x. When a zero value is detected, COMMIT skips an entire block in the matrix A. This little consideration reduces the time in the CPU version. Results showed in the Figure 5.1 were performed by using random vectors with non-zero values. Hence, Figure 5.1 shows the performance of the CPU version in the worst case.

To give a more realistic comparison between our GPU implementation and the CPU implementation, we made a serie of experiments by using 1 CPU thread. This time we varied the sparsity index of the vector x, see Figure 5.2. The sparsity index of x is the number of zero-valued elements in x divided by the size of x. For example, if the sparsity index of x is 0.1, we know that the 10% of the entries in x are zero.

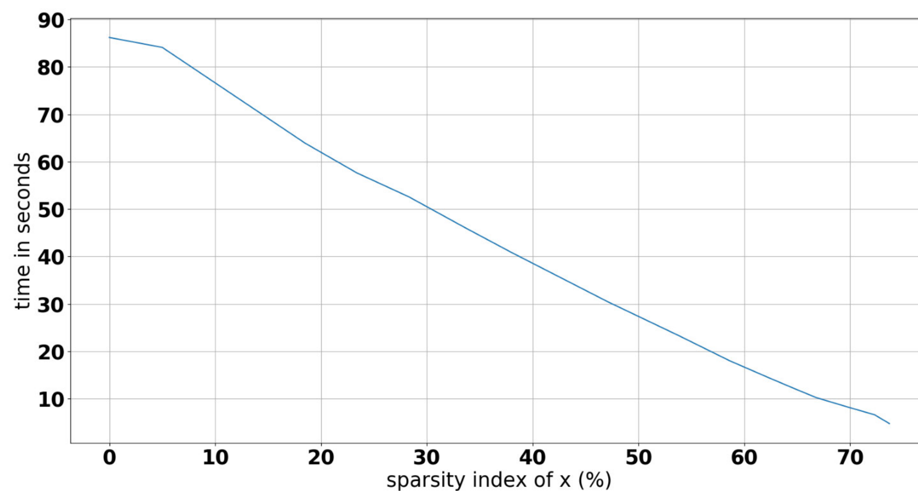

Figure 5.2: Required computational time to compute the operation Ax in CPU by varying the sparsity index of the vector x.

5.2 Operation $A^t y$

The python script used to run the experiments for the operation $A^t y$ is essentially the script we used for the operation Ax. We have to change the code from line 28 to 34 for the code below. The rest of the code remains the same. Random vectors were also used in this case.

```
1 num_tests = 50
2 for i in xrange(0, num_tests):
3     y = np.random.rand(A.shape[0]).astype('float64')
4     start  = time.time()
5     x = A.T.dot(y)
6     finish = time.time()
7     print 'Operation Atx time: ', (finish - start)
```

Measuring the time for this operation is easier because the vector y does not become sparse. We ran the CPU version by using 1, 2, 4, 8, 16 and 32 CPU threads as well. We plotted the results along with the results obtained with our GPU version in the Figures 5.3 and 5.3.

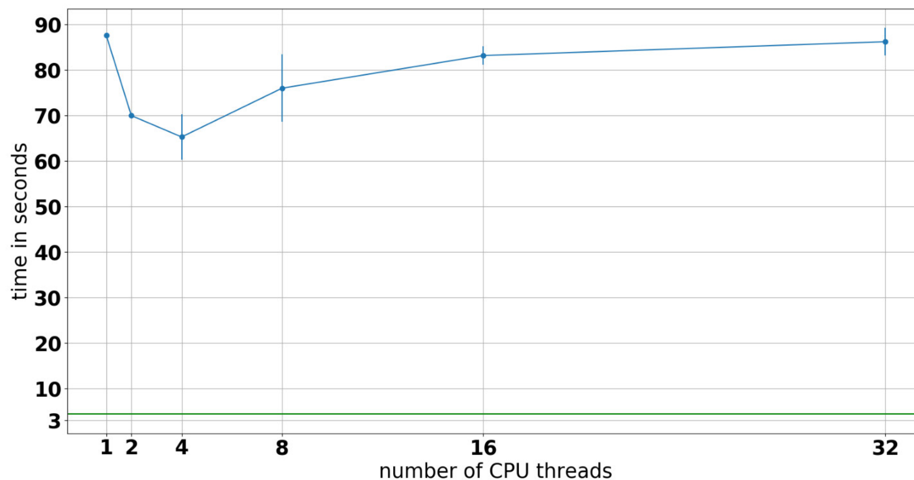

Figure 5.3: The computational time required by the operation $A^t y$ using 1, 2, 4, 8, 16 and 32 CPU threads. Horizontal axis represents the number of CPU threads and vertical axis represents the average time of 50 experiments in seconds. The blue vertical lines represent the variance of the 50 experiments. The green line shows the time of our GPU implementation in CUDA.

	Relative error: $\frac{\|x_{cpu}-x_{gpu}\|}{\|x_{gpu}\|}$
1 thread	$3.06 * 10^{-11}$
2 thread	$3.06 * 10^{-11}$
4 thread	$3.06 * 10^{-11}$
8 thread	$3.06 * 10^{-11}$
16 thread	$3.06 * 10^{-11}$
32 thread	$3.06 * 10^{-11}$

Table 5.3: Relative error between vectors x_{gpu} and x_{cpu}. The vector x_{gpu} represents the output vector obtained with the GPU version of the operation $A^t y$ and x_{cpu} represents the output vector obtained with CPU version of $A^t y$.

5.3 Model Fitting

For the model fitting performance comparisons we use real human brain data. The script to execute the model fitting is also a variation of the script presented for the operation Ax. Replacing from line 28 to the line 34 in the script for the operation Ax with the code in the box below, the model fitting can be launched.

```
1  file = open('Y', 'rb')
2  object_file = pickle.load(file)
3  Y = object_file[0]
4
5  start = time.time()
6  mit.x = commit.solvers.solve(Y, mit.A, mit.A.T, tol_fun = 1e-5,
        tol_x = 1e-6, max_iter = 10000, verbose = 1, x0 = None,
       regularisation = None)
7  end = time.time()
8
9  print "Fitting␣time␣=␣%f␣hours" % (end-start)/3600.0
10
11 mit.save_results()
12 mit.gpumem.free()
```

Finally, the results for the model fitting procedure are summarized in the Table 5.4. The fitting procedure took 2136 iterations in GPU and CPU, and the stopping criterion was the relative tolerance on the objective.

	CPU Version (1 thread)	GPU Version (CUDA)	Improved GPU Version (CUDA)
Time (hours)	65.5	7.1	5.5

Table 5.4: Computational time required to perform the model fitting with 1 CPU thread versus our implementations in GPU with CUDA.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis we have introduced a brief formulation of the COMMIT model along with some NVIDIA CUDA concepts which allow to understand the algorithms we developed. The elaboration of this thesis made the following important and remarkable contributions:

- Development of a python module which allows to manage the GPU memory from python: This python module is independent from the COMMIT framework and it provides easy tools to allocate, copy and track GPU memory. So, future users will not need CUDA programming knowledge to manage GPU memory from python.
- Homogeneous integration with the previous version: Our GPU implementation does not interfere with the already existing scripts for the COMMIT framework. People with a NVIDIA GPU can take advantage of the new GPU support, but users without NVIDIA GPUs do not have to worry about compatibility.
- Stability in the calculations: From Figure 5.1 and Figure 5.2 it can be seen that our GPU version in very stable and it is not affected by the sparsity of the vector x.
- Improved performance: Our implementation in the GPU with CUDA platform outperforms the current CPU version. We improved the fitting performance around 12 times; now the fitting procedure takes a few hours instead of some days. Operations Ax and $A^t y$ are now faster than the CPU version. This could be useful in future modifications of the COMMIT framework.

6.2 Areas of Improvement and Future Work

This thesis work had time limitations. Naturally, we have many remaining features to implement that we were not able to do because of the time limits. We have a list with the future work to do:

To UseTexture Memory: In the implementations of the operations Ax and $A^t y$, CUDA threads access to the look-up tables several times. As we saw, these look-up tables are allocated in global memory. However, you should note that we never modify the values in the look-up tables. So, we can take advantage of the texture memory. Similar to constant memory, texture memory is cached on chip [6.1]. The texture memory is read-only, thus it is faster compared to global memory and it provides higher bandwidth in special cases. Specifically, texture cache is designed for cases where threads read 2D data addresses near to each other, see Figure 6.1.

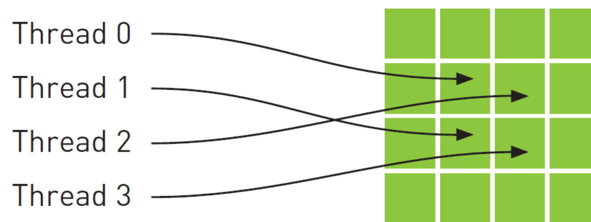

Figure 6.1: Threads accessing near addresses in a 2D texture memory. This image is property of NVIDIA Corporation.

Avoiding Bottleneck Associated with the Number of Fiber Segments per Voxel: In the section 4.6 of this document we mentioned the main bottleneck of our implementation. Voxels with the highest number of fiber segments determinate the required time to perform Ax. Figure 6.2 shows that there are just a few voxels with high amount of fiber segments and a lot of voxels with a low count of fiber segments. As a consequence, the performance of our implementation receives high impact on performance. But we could take advantage of that to design new specific kernels for the voxels with a lot of fiber segments.

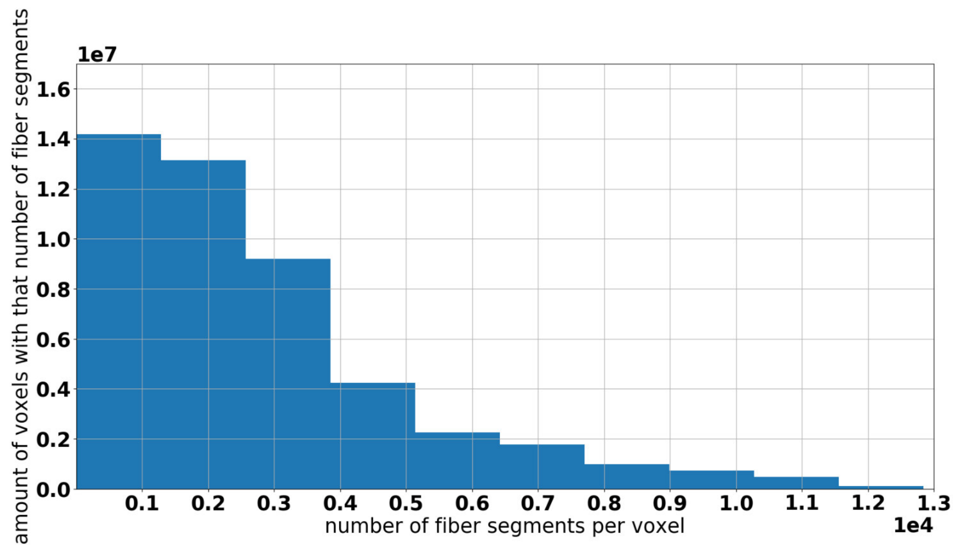

Figure 6.2: Histogram of the number of fiber segments per voxel.

Multi − GPU Support: The implementation introduced in the Section 4.4.1 calculates the signals S_v^{IC} for every $v \in \{0, \ldots, n_v - 1\}$ in parallel. These signals are independent each other, so we can distribute them into many NVIDIA GPUs to accelerate the computation of the operation Ax.

Sample Depending Kernels: In the Section 4.6 we introduced an idea which divides, in every voxel, the number of fiber segments in groups. This idea was restricted to two groups because the maximum value of n_s is 512 and we cannot launch kernels with more than 1024 threads. But with a data set where $n_s = 128$, we can create eight groups of CUDA threads in every CUDA block because $128 * 8 = 1024$ which does not exceed the threads limit. Therefore, we can design a kernel which decides how many threads groups to use by using the value of n_s.

APPENDIX A

COMPARTMENT MODELS

A.1 Intra-cellular Models

There are several models for the MRI signal associated with intra-cellular compartments. Any of these models can be used on the COMMIT framework because COMMIT is very flexible. These models are shortly explained down below:

1. The Behrens "stick" model [A.1] is a very common and computational lightweight model. This model describes diffusion in an idealised cylinder with zero radius using the sample direction $n_i \in \mathbb{R}^3$ with $1 \leq i \leq n_s$ and diffusivity coeficient $d \in \mathbb{R}_+$ as parameters. So, the entries of the signal $R \in \mathbb{R}^{n_s}$ for the model are

$$R_i = exp\left(-bd\left(n_i \cdot G\right)^2\right),\ 1 \leq i \leq n_s \tag{A.1}$$

   where $b = (\Delta - \delta/3)\left(\gamma\delta \left\|G\right\|\right)^2$. The time between the onsets of the two pulses is represented by Δ, δ is the pulse gradient duration, γ is the gyromagnetic ratio and G is the gradient vector.

2. The "cylinder" model [A.2, A.3] describes diffusion in a non-zero radius cylinder. This model is like an expantion of the previous model because it uses all of the parameters as well as an extra parameter ρ, representing a single axon radius. Each entry R_i of the signal $R \in \mathbb{R}^{n_s}$ for the model can be expressed as the product of the signals parallel and perpendicular to the cylinder axis n_i [3.5], so

$$R_i = S_i^{||} S_i^{\perp} \tag{A.2}$$

where $S_i^{||}$ represents the parallel signal and $S_i^{\perp}$ represents the perpendicular signal. Now, we shall assume that in the direction parallel to the cylinder water particules move freely. Then $S_i^{||}$ is the same signal as A.1. For the signal $S_i^{\perp}$ we can consider this generalized form

$$S_i^{\perp} = exp\left(L\left(G \cdot G - (G \cdot n_i)^2\right)\right) \tag{A.3}$$

where

$$L = -2\gamma^2 \sum_{m=1}^{\infty} \frac{2d\beta_m^2\delta + 2Y(\delta) + 2Y(\Delta) - Y(\Delta - \delta) - Y(\Delta + \delta) - 2}{d^2\beta_m^6\left((\alpha/2)^2\beta_m^2 - 1\right)}. \tag{A.4}$$

The value β_m is the mth root of the equation $\frac{\partial J}{\partial \beta}(\beta\alpha/2) = 0$ where $\frac{\partial J}{\partial \beta}$ is the first order derivate of the Bessel function and $Y(x) = exp\left(d\beta_m^2 x\right)$.

3. The third model is called "GDRCylinders" and it is basically the same cylinder model. But this model uses a gamma-distributed radius according to [3.2]. So, instead of the single parameter $\rho \in \mathbb{R}$, we have a random variable $\rho \sim Gamma(\alpha, \beta)$. As parameters for this distribution we have the shape parameter α and the scale parameter β. Therefore, the density function of ρ is given by

$$f_\rho(r; \alpha, \beta) = \frac{r^{\alpha-1} exp(-r/\beta)}{\Gamma(\alpha)\beta^\alpha}. \tag{A.5}$$

Please note that $\alpha\beta$ is the mean and $\alpha\beta^2$ is the variance.

BIBLIOGRAPHY

[1.1] Emanuele Olivetti; Nusrat Sharmin; Paolo Avesani. Alignment of Tractograms As Graph Matching. Frontiers in Neuroscience: Brain Imaging Methods. 2016.

[1.2] Jun Zhang; Hao Ji; Ning Kang; Ning Cao. Fiber Tractography in Diffusion Tensor Magnetic Resonance Imaging: A Survey and Beyond. Department of Computer Science, University of Kentucky. 2005.

[1.3] Roland Bammer. Basic Principles of Diffusion -Weighted Imaging. European Journal of Radiology. 2002.

[1.4] Peter J. Basser; James Mattiello; Denis LeBihan. MR Diffusion Tensor Spectroscopy and Imaging. Biophysical Journal Volume 66. 1994.

[1.5] Christopher R. Madan. Creating 3D visualizations of MRI data: A brief guide. US National Library of Medicine; National Institutes of Health. 2015.

[1.6] Y. Rathi; J Malcolm; S. Bouix; C-F. Westin; M. E. Shenton. False Positive Detection using Filtered Tractography. Proc. Intl. Soc. Mag. Reson. Med. 18. 2010.

[1.7] Peter J. Basser; Sinisa Pajevic; Carlo Pierpaoli; Jeffrey Duda; Akram Aldroubi. In Vivo Fiber Tractography Using DT-MRI Data. Magnetic Resonance in Medicine 44:625-632. 2000.

[1.8] Ben Jeurissen; Maxime Descoteaux; Susumu Mori; Alexander Leemans. Diffusion MRI Fiber Tractography of the Brain. NMR in Biomedicine. 2017.

[1.9] Hugues Duffau. Diffuse Low-grade Gliomas in Adults. Page 393. Springer. 2017.

[1.10] https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190

[1.11] https://www.sciencedaily.com/terms/axon.htm

[2.1] https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[2.2] https://www.quora.com/What-is-GPU-and-CPU-differences-and-similarities

[2.3] https://hardzone.es/2018/04/22/memoria-ram-ddr-vs-gddr-diferencias/

[2.4] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__UNIFIED.html

[2.5] Jason Sanders; Edward Kandrot. CUDA By Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley. NVIDIA Corporation. 2011.

[2.6] John Cheng; Max Grossman; Ty McKercher. Professional CUDA C Programming. Wrox; NVIDIA Corporation. 2014.

[2.7] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[2.8] https://www.quora.com/What-is-the-difference-between-GDDR-and-DDR-memory

[2.9] https://www.quora.com/Why-dont-GPUs-have-branch-predictors

[3.1] Alessandro Daducci; Alessandro Dal Palu; Alia Lemkaddem; Jean-Philippe Thiran. COMMIT: Convex Optimization Modeling for Micro-structure Informed Tractography. IEEE Transactions on Medical Imaging, 2013.

[3.2] Eleftheria Panagiotaki; Torben Schneider; Bernard Siow; Matt G. Hall; Mark F. Lythgoe; Daniel C. Alexander. Compartment models of the diffusion MR signal in brain white matter: A taxonomy and comparison. Elsevier Inc., 2011.

[3.3] C. H. Neuman. Spin echo of spins diffusing in a bounded medium. Chevron Oil Field Research Company, 1973.

[3.4] Behrens; T.E.J.;Woolrich; M.W.; Jenkinson; M.; Johansen; H. Characterization and propagation of uncertainty in diffusion-weighted MR imaging. Magn. Reson. Med., 2003.

[3.5] C. H. Neuman. Spin-echo of Spins Diffusing in a Bounded Medium. Chevron Oil Field Research Company. 1973.

[3.6] Assaf; Y.; Freidlin; R.Z.; Rohde; G.K.; Basser; P.J. New modeling and experimental framework to characterize hindered and rectricted water diffusion in brain white matter. Magn. Reson. Med., 2004.

[3.7] J.-D. Tournier; F. Calamante; A. Connelly. Robust Determination of the Fibre Orientation Distribution in Diffusion MRI: Non-negativity Constrained Super-resolved Spherical Deconvolution. NeuroImage, Vol. 35. 2007.

[3.8] N. S. White; T. B. Leergaard; H. D'Arceuil; J. G. Bjaalie; A. M. Dale. Probing Tissue Microstructure with Restriction Spectrum Imaging: Histological and Theorical Validation. Hum Brain Mapp, Vol 34, No. 2. 2013.

[3.9] C. L. Lawson; R. J. Hanson. Solving Least Squares Problems. Society for Industrial and Applied Mathematics Vol. 161. 1974.

[3.10] Y. Saad. Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics. 2013.

[3.11] Amir Beck; Marc Teboulle. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. Society for Industrial and Applied Mathematics. 2009.

[4.1] Nathan Bell; Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Corporation. 2008.

[4.2] https://github.com/daducci/COMMIT

[4.3] https://docs.python.org/3/library/ctypes.html

[4.4] Mark Harris. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology

[4.5] https://en.wikipedia.org/wiki/Sparse_matrix

[6.1] http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[A.1] Behrens T.E.J.; Woolrich M.W.; Jenkinson M.; Johansen H. Characterization and Propagation of Uncertainty in Diffusion-weighted MR Imaging. Magn. Reson. Med. 1077-1088. 2003.

[A.2] Alexander D.C. A General Framework for Experiment Design in Diffusion MRI and its Application in Measuring Direct Tissue-microstructure Features. Magn. Reson. Med. 439-448. 2008.

[A.3] Alexander D.C.; Hubbard P.L.; Hall M.G.; Moore E.A.; Ptito M.; Parker G.J.M.; Dyrby T.B. Orientationally Ivariant Indices of Axon Diameter and Density from Diffusion MRI. Neuroimage 52, 1374-1389. 2010.

[A.4] C. H. Neuman. Spin-echo of Spins Diffusing in a Bounded Medium. Chevron Oil Field Research Company. 1973.