Structure Tensor Tractography

Manoj Kumar Cebol Sundarrajan
1620546

INTRODUCTION

Data Sets acquired using high throughput imaging methods are dense and very large, making them difficult to segment, analyze and visualize. A slice of data of a kidney obtained from the microscopic method is shown in figure 1.

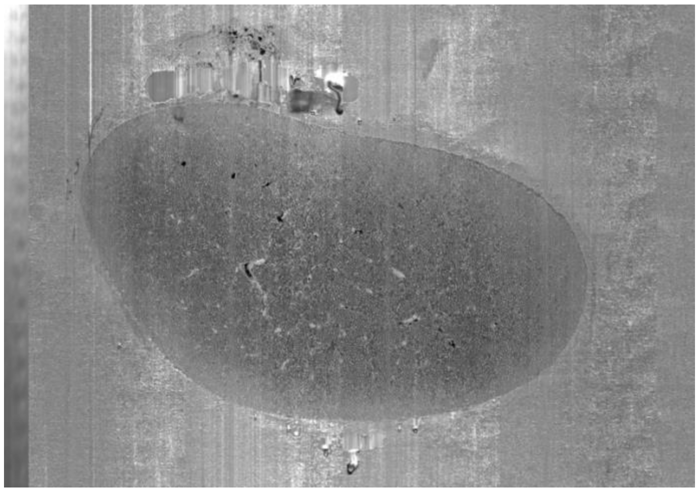


Fig.1 A dense image of a kidney obtained from high throughput microscopic imaging method

The method I will be using to visualize this large dataset is a technique presented by Srijani Mukherjee from University of Houston under the guidance of Dr. David Mayerich. The method proposed by Srijani Mukherjee is to construct the structure tensor field, which can be build out of core for large datasets. The processed data consists of a tensor field describing the local structure at each point in the data set. Then we calculate the Eigen vectors at each point., which shows the direction of the fibers. Then we use tractography to visualize the fibers. We use Euler's method of integration to update the position at each timestep. We will work on a subset of the image. A slice of the subset of the image is shown in figure 2.

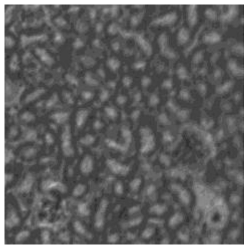


Fig.2 Sub image of kidney obtained from high throughput microscopic imaging method

Steps:

1. Read the data

Read the pixel data in Jpeg form. I will be using an external library CImg to achieve this task.

2. Calculating partial derivatives

Forward Difference Formula:

$$d\frac{(pixel)}{dx} \cong \frac{pixel(x+1) - pixel(x)}{h}$$

Forward Difference Formula:

$$d\frac{(pixel)}{dx} \cong \frac{pixel(x) - pixel(x-1)}{h}$$

Center Difference Formula:

$$d\frac{(pixel)}{dx} \cong \frac{pixel(x+1) - pixel(x-1)}{2 * h}$$

Where h is the step size.

This same formula can be applied to y and z to calculate partial derivates along y and z direction. Forward and backward difference formula is used on the edges. Centered difference formula is used at all other points.

2. Calculating structure tensor.

Using the derivates calculated, calculate 3 * 3 structure tensor at each point using the formula provides below

$$Tensor(x, y, z) = \begin{bmatrix} dx * dx & dx * dy & dx * dz \\ dy * dx & dy * dy & dy * dz \\ dz * dx & dz * dy & dz * dz \end{bmatrix}$$

3. Apply blurring

$$\frac{\mathrm{T}[1] \; + \; \mathrm{T}[2] + \cdots + \mathrm{T}[\mathrm{filter\_size}]}{filter\_size}$$

Box filter is separable and is applied along all the three direction.

4. Calculate the Eigen vectors

5. Apply tractograpghy

Euler's method:

$$pos\begin{bmatrix} x(n+1) \\ y(n+1) \\ z(n+1) \end{bmatrix} = pos\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \Delta t * slope\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\boldsymbol{T}(\boldsymbol{n}+\mathbf{1}) = \boldsymbol{T} + \Delta \boldsymbol{t}$$

The slope is given by the eigen vectors. ΔT is the timestep.

Calculate the points using Euler's integration and write the trace to a file.

Output:

The output of tractography is shown in figure 3.

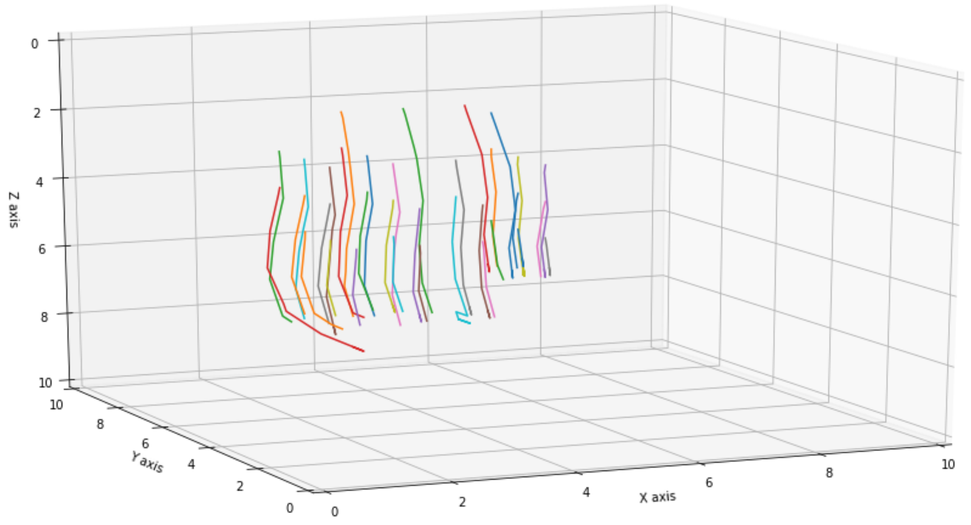


Fig 3. The output of tractography for few starting points plotted using python

CPU Implementation

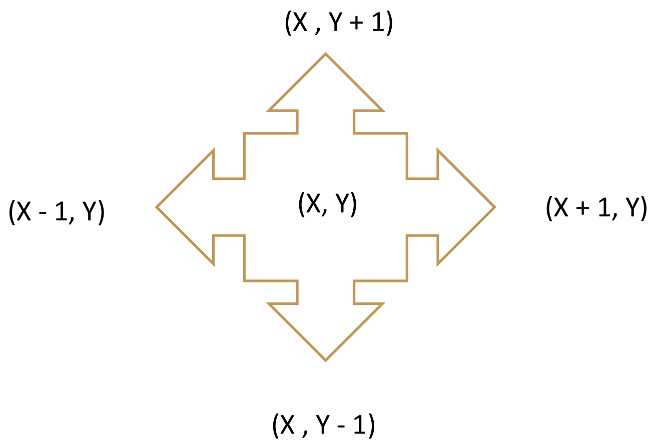
The implementation of each step is explained below.

1. Reading the data

The data is present in the form of Images stored in Jpeg format. I am using an external library called CImg (http://cimg.eu) to read the data. The read has to be sequential and the images has to be read one by one. There is no parallelization in this part of the code. So, this section of the code cannot be offloaded to a GPU.

2. Calculating Partial derivatives

Calculating the derivatives at each point is a highly data parallel problem. CPU does it in sequential order. To calculate derivate of each point you will typically only need the data of adjacent cells. This part of the code is scalable on the GPU.



3. Calculating structure tensors

This is an embarrassingly parallel problem. Each point is dependent on the derivates at that point. This part should should scale extremely well on a GPU.

4. Appling blurring to the tensors

I'm using a uniform filter to blur the image. The filter size is 16 along the X, Y direction and 4 along the Z direction. This is a computationally intensive and highly parallelizable section that can be offloaded to a GPU.

5. Calculating Eigen vectors

For simplicity, I decided to go with an external library Eigen (http://eigen.tuxfamily.org/). Reason I chose this library is I was not able to find any analytical solution to calculate the Eigen vector of matrices. Most of the solutions included solving of linear equations which is iterative, time consuming and harder to implement. This is the most time-consuming portion of the code. It's an embarrassingly parallel problem. It should have had the most significant speedup on the GPU. Unfortunately, the Eigen datatypes were not fully supported in nvcc compiler (https://eigen.tuxfamily.org/dox/TopicCUDA.html) and was unable to offload to GPU

6. Tractography

The file step of the algorithm consists of tracing a point along the vector direction using Euler's method of integration. This is an embarrassingly task parallel problem as Ideally, we can start to trace starting from each point in the 3D image. Doing this creates, millions of traces for each point, which requires efficient database to handle the files. The filesystem started to choke in bridges when I tried. So, I had to restrict myself to few hundreds of points, I recommend using 5 * 5 * 5 staring points. The traces are imported to python and visualized using matplotlib and mplot3D.

PARALLEL IMPLEMENTATION

KERNEL 1: CALCULATING PARTIAL DERIVATIVES

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
Calculate the Overall 1D index
If (threads within bounds)
        If (X == 0) dx[1Dindx] = (pixel[1D_index+1] – pixel[1D_index])/h
        If (X == width) dx[1Dindx] = (pixel[1D_index] – pixel[1D_index-1])/h
        else dx[1Dindx] = (pixel[1D_index+1] – pixel[1D_index-1])/2*h
        …
        …
        Apply same algorithm for Y and Z direction
Else return the threads
```

Profiling result:

The time taken to run this kernel was 1.6585 ms. This kernel had a speed up of 586 times over the CPU code

Looking at occupancy using the nvprof command

```
nvprof --kernels "partial_derv_gpu"  --metrics "achieved_occupancy" ./gpu
```

```
Device "Tesla P100-PCIE-16GB (0)"
    Kernel: partial_derv_gpu(float*, float*, float*, int*, int, int, int, int, int)
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
1	achieved_occupancy	Achieved Occupancy	0.832636	0.832636	0.832636

KERNEL 2 : CALCULATE THE TENSOR

Pseudocode:

```
Calculate the index
If (index with bounds)
        T[index].a1 = dx[index] * dx[index]
        T[index].a2 = dx[index] * dy[index]
        T[index].a3 = dx[index] * dy[index]
        T[index].b2 = dy[index] * dy[index]
        T[index].b3 = dy[index] * dz[index]
        T[index].c3 = dz[index] * dz[index]
    else return threads
```

Profiling result:

Time taken to calculate tensors on GPU was 5.16634 ms. This is 206 times faster than the CPU code.

Looking at occupancy using the nvprof command

nvprof --kernels "partial_derv_gpu" --metrics "achieved_occupancy" ./gpu

```
Device "Tesla P100-PCIE-16GB (0)"
  Kernel: tensor_gpu(matrix*, float*, float*, float*, int)
Invocations        Metric Name              Metric Description       Min        Max        Avg
     1          achieved_occupancy          Achieved Occupancy    0.721400   0.721400   0.721400
```

Kernel 3 : Blurring the Tensors

This part of the code involved 5 steps.

Step 1: Apply padding to the data

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
Calculate the Overall 1D index
If(index within bounds)
    Calculate local_X_index
    Calculate local_Y_index
    Calculate local_Z_index
    Calculate local_1D_index
    if(indices within the require area)
        Copy data to padded array
    else
        make the halos zero
else return the threads
```

Profiling result:

This section took 11.8064 ms. A speed up of 89 times over CPU code. The occupance reported from profiling was 0.867613.

Step 2: Apply Blurring along X direction

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
If(threads within bounds)
            Copy required data to shared memory
synch the threads within block
if(threads within required bounds)
            Compute shared index
             for i from 0 to filter size
                        accumulate the tensors values along x direction
                        normalize with filter size
else return the threads
```

Profiling result:

Time taken to complete this kernel was 9.14461 ms. The speedup was tremendous 931 times due to the usage of shared memory.

The average occupancy is 0.236182 which is low but the shared memory efficiency was 98.19 percent. The result was achieved using the following command below

```
nvprof --kernels "filter_x_gpu"  --metrics "sm_efficiency" ./gpu
```

Device "Tesla P100-PCIE-16GB (0)"

Kernel: filter_x_gpu(matrix*, matrix*, int, int, int, int, int, int, int)

Invocations	Metric Name	Metric Description	Min	Max	Avg
1	sm_efficiency	Multiprocessor Activity	98.19%	98.19%	98.19%

Step 3: Apply Blurring along Y direction

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
If(threads within bounds)
            Copy required data to shared memory
synch the threads within block
if(threads within required bounds)
            Compute shared index
             for i from 0 to filter size
                        accumulate the tensors values along Y direction
                        normalize with filter size
else return the threads
```

Profiling result:

Time taken to complete this kernel was 8.7159 ms. The speedup was tremendous 1015 times due to the usage of shared memory.

The average occupancy is 0.235489 which is low but the shared memory efficiency was 98.09 percent.

Step 4: Apply Blurring along Z direction

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
If(threads within bounds)
    Copy required data to shared memory
synch the threads within block
if(threads within required bounds)
    Compute shared index
     for i from 0 to filter size
        accumulate the tensors values along Z direction
        normalize with filter size of Z
else return the threads
```

Profiling result:

Time taken to complete this kernel was 5.9952 ms. The speedup was tremendous 1467 times due to the usage of shared memory.

The achieved occupancy is 0.666987 and the shared memory efficiency was 99.93 percent.

Step 5: Un pad the data

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
Calculate the Overall 1D index
If(index within bounds)
    Calculate local_X_index
    Calculate local_Y_index
    Calculate local_Z_index
    Calculate local_1D_index
    if(indices within the require area)
        Copy data to un padded array
else return the threads
```

Profiling result:

This section took 11.5745 ms. A speed up of 454 times over CPU code. The occupance reported from profiling was 0.875693.

KERNEL 4 : TRACTOGRAPHY

Pseudocode:

```
Calculate Index along X direction
Calculate Index along Y direction
Calculate Index along Z direction
Calculate the Overall 1D index
If(indices within bounds)
    Call Euler's method device function with the thread indices
else return the threads
```

Euler's method:

Pseudocode:

```
Assign initial using the thread indices
copy the indices to trace
Initialize step = 0
While (Step  < Num_steps)
    Calculate 1D index of Initial position
    Calculate new positon using Euler's formula
    If( All positions within boundary)
        Calculate the 1D index of new position
        Compute the dot product of new and old vector
        if(dot product is negative)
            Recompute the new position with negative indices of old position
        increment the step
        Add new position to the trace
        Update initial position with new position
    else
        Increment the step
        update trace with out of bound indices.
return
```

Profiling result:

The achieved occupancy is 0.139140. This is very low because I'm launching only 125 threads. There will be a significant increase in occupancy and speedup over CPU code if millions of threads are launched.

TIMING AND BOTTLENECKS

Timing Results for CPU:

93 files

time taken to read files	: 2334.12 ms
time taken to calculate partial derivatives	: 958.07 ms
time taken to calculate Tensor field	: 1065.36 ms
time taken to apply Padding	: 989.101 ms
time taken to apply blur along x axis	: 8522.31 ms
time taken to apply blur along y axis	: 8853.27 ms
time taken to apply blur along z axis	: 3004.31 ms
time taken to remove padding	: 5260.06 ms
time taken to calculate eigen vector	: 30.2457 s (OpenMP 16 threads)
time taken to do cryptography and writing it	: 2.85251 s (OpenMP 16 threads)

Timing Results for GPU:

93 files

time taken to read files	: 3132.39 ms	CPU (using Cimg Library)
time taken to calculate derivates	: 1.6585 ms	GPU
time taken to calculate tensors	: 5.16634 ms	GPU
time taken to apply padding	: 11.8064 ms	GPU
time taken to apply blurring along x direc	: 9.19501 ms	GPU-SHMEM
time taken to apply blurring along y direc	: 8.7159 ms	GPU-SHMEM
time taken to apply blurring along z direc	: 5.9849 ms	GPU-SHMEM
time taken to remove padding	: 11.512 ms	GPU
computing Eigen vectors accelerated using OpenMP		
time taken to calculate eigen vector	: 30203.1 ms	CPU-OPENMP(Used 16 threads)
time taken to do tractography	: 69.4649 ms	GPU

The major bottle necks in the CPU implementation are

1. Blurring the Image

2. Calculating the Eigen vectors

3. tractography

Using GPU All the three bottlenecks can be clearly eliminated. Unfortunately, I was not able to Implement the calculation of Eigen vectors in GPU. The Eigen library datatypes were not supported by GPU.

PITFALLS AND CHALLENGES

I had 2 pitfalls while developing the GPU code.

1. I wanted to implement the Partial derivatives calculation using shared memory. Then I realized that the data had to be padded over the edges. Also, the edges have to be handled carefully. I did not have enough time to implement this.

2. I was not able to implement the calculation of Eigen vector on a GPU even though it's a straight forward problem. Turns out the eigen vector calculation is the most time-consuming part in the CPU implementation.

Compiling And Running

Compiling and running instructions can be found at this Github link under README.

https://github.com/manoj1511/ECE_6397-GPU_Programming/tree/master/final_project