Efficient fMRI Analysis and Clustering on GPUs

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State University

By

Dharneesh Talasu, B.E

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Thesis Committee:

Dr. Raghu Machiraju, Advisor
Dr. Gagan Agrawal
ABSTRACT

*Graphics processing units (GPUs)* traditionally have been used to accelerate only parts of the graphics pipelines. The emergence of the new age GPUs as highly parallel, multi-threaded and many core processor systems with the ability to perform general purpose computations has opened doors for new form of heterogeneous computing where the GPU and CPU can be used together in accelerating the underlying computations. *General-purpose computing on graphics processing unit* (GPGPU, also referred to as GP\(^2\)U) techniques can be used to perform highly data parallel computations and to accelerate some critical sections of an application. Accelerating the computation of fMRI analysis on a graphics processing unit is mainly attractive when used in a clinical environment.

In this thesis, I discuss methods which try to exploit the capabilities provided by GPUs to accelerate the analysis of time varying data acquired during fMRI experiments for identifying regions of activity/inactivity. Static activation maps are obtained by inspecting voxels independently with the help of statistical methods in parallel using CUDA (*Compute unified device architecture*) threads. I provide an efficient strategy for mapping each individual time varying voxels to GPU kernel threads for data parallel analysis of fMRI data and present GPU version of methods used in the fMRI analysis pipeline based on voxel to thread mapping technique. Also, an efficient method for *octree* based hierarchical clustering of voxels on a GPU and using
a combination of GPU and CPU for enhanced clustering speedup is discussed. A comparison between the data parallel methods implemented on GPU and the corresponding CPU implementations and overall speed up achieved using combined GPU and CPU implementations in octree based hierarchical clustering is discussed.
Dedicated to my parents and sisters
ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge and thank the invaluable support and guidance of my advisor, Dr Raghu Machiraju. I am deeply indebted to him and express my gratitude for his support, guidance and advice. This has been a privilege to work with him and I got to learn a lot from him during this time.

I would like to acknowledge and thank Dr. Gagan Agrawal for agreeing to be a part of my thesis defense committee.

I would like to acknowledge and thank Dr. Janoos for providing me with invaluable support and guidance and also Dr. Irfanoglu for guiding me during the initial stages and for helping me understand MRI and DTI.

I do not have enough words to express my deep gratitude towards my parents and my sisters. Nothing that I will ever do can even come close to their love and support for me and the sacrifices they have made to see me here.

Finally, I thank all my friends at OSU for sharing this journey and making it a memory to be cherished forever.
VITA

January 12, 1982 ....................... Born - Andhra Pradesh, India

2005 ........................................... B.E., Information Technology.,
West Bengal University of Technology,
Kolkata, India.

Jan 2010- Sep 2010 ....................... Graduate Research Associate,
The Ohio State University

Dec 2007- Aug 2009 ....................... Senior Software Engineer,
Patni Computer Systems., Pvt Ltd, India

Feb 2006- Nov 2007 ....................... Software Engineer,
Patni Computer Systems., Pvt Ltd, India

PUBLICATIONS

Research Publications


FIELDS OF STUDY

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background of the problem</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Research Statement</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Principal Contribution</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Outline of Solution</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Organization of Thesis</td>
<td>4</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 MRI/fMRI - A Brief Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2 GPU Architecture - An Overview</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 NVidia GPU Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.3 CUDA Programming model</td>
<td>10</td>
</tr>
</tbody>
</table>

vii
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1   Sample NVidia GPU Hardware</td>
<td>11</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Schematic plot for a BOLD response</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Example fMRI data with regions of activation</td>
<td>7</td>
</tr>
<tr>
<td>2.3 GPU vs. CPU</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Nvidia GPU Architecture</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Grid &amp; Block Structure.</td>
<td>13</td>
</tr>
<tr>
<td>3.1 fMRI analysis and clustering flowchart</td>
<td>15</td>
</tr>
<tr>
<td>3.2 fMRI data layout in memory as an array.</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Sample octrees for 3D objects of side lengths 1 and 2</td>
<td>23</td>
</tr>
<tr>
<td>3.4 A sample computation of SFC Index from a given ((x, y, z)) coordinate</td>
<td>26</td>
</tr>
<tr>
<td>3.5 Linearizing and sorting of data based on Z-SFC Index</td>
<td>27</td>
</tr>
<tr>
<td>3.6 Cluster structure format</td>
<td>32</td>
</tr>
<tr>
<td>3.7 Shuffling of voxels in thread local array to group significant and insignificant voxels separately.</td>
<td>32</td>
</tr>
<tr>
<td>3.8 Hierarchical clustering on GPU, with mechanism to overcome the GPU constraints.</td>
<td>38</td>
</tr>
<tr>
<td>3.9 Cluster explorer GUI</td>
<td>39</td>
</tr>
</tbody>
</table>
Listings

2.1 Sample Matrix Addition kernel . . . . . . . . . . . . . . . . . . . . . . 12
3.1 Sample division of blocks and grids . . . . . . . . . . . . . . . . . . . . 19
3.2 Sample kernel with \textit{thread ID} to array mapping . . . . . . . . . . . . . 19
3.3 Sample code to create vectors of intensities for each voxel . . . . . . . . 21
# List of Algorithms

1. Octree based Hierarchical Clustering (CPU version) ................................................. 25
2. Significant Voxel Filtering and Initialization of Clusters at Leaf Level of Octree .................................................................................................................. 30
3. Agglomerative Hierarchical Clustering based on \textit{Octree Model} .......................... 31
CHAPTER 1

INTRODUCTION

This thesis discusses General-purpose computing on graphics processing units (GPGPU) based implementation of fMRI analysis. The analysis algorithms used in this thesis for accelerating on GPUs are based on CPU versions of algorithms presented in “Visual analysis of brain activity from fmri data”, Firdaus et al [7]. The introduction opens by a discussion on the background and the proposed solution in Section 1.4. Finally, the organization of the remaining of this thesis is laid out in Section 1.5.

1.1 Background of the problem

Functional magnetic resonance imaging (fMRI), is one of the most popular neuroimaging tools which contains spatial as well as temporal information. Temporal resolution is measured in seconds. This form of imaging method is based on nuclear magnetic resonance (NMR) signal changes detected due to changes in levels of oxygenation of blood which is otherwise called as blood-oxygen-level dependence (BOLD) signals in the regions of activations. Statistical analysis techniques can be applied on the acquired fMRI for localization of different functional domains of the brain.

Data from experiments comprises of series of 3D volumes obtained over a duration (seconds). Each voxel in the 3D volume needs to be tested by suggested statistical
methods [7] over the entire time series to detect changes related to activity/inactivity and voxels with similar behavior need to be clustered together based on the suggested dissimilarity metric [7]. The number of voxels that need to be analyzed is of the order of $O(N^3M)$ where ‘N’ is the size of each dimension and ‘M’ being the number of volumes in the time series. Due to the requirement for testing these large number of voxels for activity, the execution of analysis algorithms on traditional CPUs are time consuming due to their nature of sequential execution. The analysis can be drastically accelerated using GPGPU programming model by exploiting their feature to support data parallel algorithms.

Clustering of voxels based on their similarity in value according to a distance metric [7] with other neighboring voxels/clusters is a major time consuming step during the execution of the pipeline. To improve the execution time, use of octree based clustering has been suggested in the work Firdaus et.al [7] [3]. This clustering algorithm for identifying VOI can be further accelerated by parallelizing the construction of octree on a GPU.

1.2 Research Statement

In this thesis, I introduce fMRI analysis and hierarchical clustering methods which are based on GPGPU model using GPU instead of traditional CPU and investigate the following:

“(1) fMRI analysis: Can GPGPU techniques be applied towards the analysis of fMRI data given the large volumes of data? How to achieve maximum throughput using GPU? Can a GPU outperform the CPU in the analysis and how do they compare in performance?”
“(2) **Hierarchical clustering.** Can GPGPU techniques be applied in hierarchical clustering of large data and large number of subjects? What is the performance gain achieved by using GPU instead of CPU? Can a combination of the GPU and CPU be used to achieve optimal performance? How do CPU, GPU and a combination of the GPU and CPU compare in performance in clustering?”

### 1.3 Principal Contribution

In this thesis, I present methodologies to accelerate the analysis of fMRI time series volumes using GPGPU techniques. A significant contribution is, an algorithm for GPGPU based **parallelized single-linkage agglomerative hierarchical clustering** of voxels in a fMRI brain volume. The distance metric used for clustering is Mahalanobis distance [9] based dissimilarity metric discussed in [7].

### 1.4 Outline of Solution

In order for the analysis of the fMRI data, I consider three phases in the pipeline:

1. In the first phase, a set of brain responses are computed for different hemodynamics. This can be obtained by 1D-convolution of the stimulus function (which specifies the experimental task given by the onset and duration of the stimulus) with a set of typical HRFs. This step returns the features of interest from the experimental task. The execution of this step is left to the CPU, since the amount of time required for this phase is of the order of milli seconds and the latency in data transfer between the CPU (host) and GPU (device) can overshadow the acceleration achieved by GPU.
2. Phase II involves statistical analysis of fMRI data. Statistical computation methods involved in this phase are data parallel algorithms. For parallelization in this phase, CUDA (Compute unified device architecture) kernel threads equal to the number of voxels in the 3D volume are created and the analysis methods for the observed signal over the experiment duration at each voxel position is executed in parallel.

3. In the final phase, voxels are clustered based on a dissimilarity metric proposed by Firdaus et.al in [7]. Parallelized version of this contributes significantly in the reduction of overall analysis time of fMRI data and creation of VOIs.

1.5 Organization of Thesis

The remaining of this thesis is organized as follows:

In chapter 2, Section 2.1 provides a background of MRI and fMRI. Then in Section 2.2, I provide a background of GPUs, GPGPUs and CUDA architecture and programming model. Chapter 3 deals with various statistical computations used in the analysis of fMRI, the parallelization model and the parallelization of hierarchical clustering based on octree data structure. A comparison of CPU vs. GPU vs. GPU+CPU execution timings is provided in 3.5. Chapter 4 is dedicated for conclusion and future work.
CHAPTER 2

BACKGROUND

This chapter is intended to provide an overview of fMRI, GPUs and CUDA architecture. It is intended to highlight the key concepts fMRI and CUDA and set the context for the later sections of the document.

2.1 MRI/fMRI - A Brief Overview

*Functional magnetic resonance imaging* (fMRI) is a *magnetic resonance image* (MRI) which has *nuclear magnetic resonance (NMR)* as it’s underlying principle. NMR is based on Zeeman effect [18], in which spectral lines are split into several components in the presence of strong magnetic field. The effect is due to the distortion of electron orbitals in presence of a strong magnetic field. This lead to the development of MRI in which tomographic images of brain are generated using NMR based methods [8]. In the case of an MRI, tomographic images of brain are generated by measuring the NMR signal for the Hydrogen isotope (\(^1\)H) which is present in abundance in brain tissue manifest as water.

In the earliest experiments of fMRI, measurement of functional activation using an MRI scanner was based on the changes of blood volume level w.r.t activations. Contrast agents such has gadolinium-DTPA have been used to alter the relaxation of
blood water and tissue water after an activation. Time courses of the signal changes induced by the contrast agent are measured using the relative differences of blood volumes in activation regions with that of inactive regions. The later discovery of blood-oxygen-level dependence (BOLD) [11] by Seiji Ogawa in 1990 at AT&T Bell Labs has superseded contrast agent based fMRI experiments and formed the basis for the present day fMRIs based on BOLD and the method is known as BOLD fMRI. BOLD fMRI exploits the magnetic property of Hemoglobin which is diamagnetic when oxygenated (oxyhemoglobin) and paramagnetic when de-oxygenated (deoxyhemoglobin). BOLD signal intensities increase with increase in concentration of oxyhemoglobin which results in differences in NMR signal for different levels of oxygenation of blood. By collecting the MRI image data over the duration of the experiment, changes in BOLD contrast can be measured as shown in Figure 2.1 which illustrates BOLD response for a stimulus function.

![Figure 2.1: Schematic plot showing the time course of BOLD response for an experimental stimulus](image)

Figure 2.1: Schematic plot showing the time course of BOLD response for an experimental stimulus
In case of BOLD fMRI, BOLD contrast changes are measured by rapid acquisition of a sequence of $T_2^*$ MRI images using *echo-planar imaging* (EPI) methods. Images can be acquired with moderately good spatial and temporal resolution and the frequency at which images are captured varies from 1-4 seconds. The time varying changes in intensity values at a given voxel location along the series of volumes acquired is used in the statistical analysis of fMRI to identify regions of activation. The example in Figure 2.2 shows fMRI data with regions of activation.

![Sample fMRI data with regions of activation](image)

Figure 2.2: Sample fMRI data with regions of activation activations shown in yellow and red.

### 2.2 GPU Architecture - An Overview

The term *graphics processing unit* (GPU) was first defined by NVidia with their release of NVidia GeForce 256, a single chip processor, which is considered to be the
world’s first GPU. A GPU is designed to accelerate graphics applications using their high parallel processing capability, which is of the order of 100's of Gflops/sec. The high rates of floating point operations are possible due to the specialized design of GPU in which majority of the transistors are dedicated for data processing and fewer transistors dedicated for data caching and flow control. This design of GPU facilitates the execution of same program on multiple data (SPMD - Single Program Multiple Data) elements in parallel on multiple threads and thereby achieving higher arithmetic to memory operation ratio compared to CPU. A schematic illustration of GPU vs. CPU is provided in Figure 2.3.

![Figure 2.3: GPU vs. CPU: More transistors for data processing and fewer transistors for flow control & caching](image)

Applications with requirement to process large data sets, where same algorithm is applied on each element in the data set, can be parallelized using a Data Parallel model that can executed on a GPU. Each data element from the data set can be mapped to a GPU thread for concurrent execution. In the case of fMRI data, each voxel and its time varying data can be mapped to a thread on GPU. The statistical
analysis algorithms are executed for each voxel time series data in parallel, which facilitates manifold increase in performance of the analysis.

The time taken for data transfer between host (CPU) and device (GPU) can be ignored when compared with speedup achieved with data parallelism.

2.2.1 NVidia GPU Architecture

Any NVidia GPUs device consists of an array of multi-threaded streaming multi-processors (SMs). An SM is designed to execute hundreds of threads concurrently for which it employs single instruction multiple threads (SIMT) architecture. Instruction level parallelism is achieved by pipelining within a thread and thread level parallelism is achieved by hardware multi-threading. Each SM consists of 8 streaming processors (SPs) or processor cores, with each core running at 1.3 GHz. In a an NVidia GPU, the SM is responsible for the creation and management of the threads. SM executes threads in groups of 32 threads known as warps. More details on SIMT architecture and hardware multi-threading can be obtained from CUDA C programming guide [10].

NVidia GPUs have multiple memory levels:

1. Global memory (off-chip)
2. Shared memory (on-chip)
3. Constant memory (on-chip cache)
4. Texture memory
5. Local memory
The block diagram in Figure 2.4 shows the NVidia GPU architecture with various memory levels and their access levels. Global memory can be accessed by SMs as well as SPs. Shared memory is local to a SP. Executing threads have their local memory in the processor registers.

![Nvidia GPU Architecture](image)

Figure 2.4: Nvidia GPU Architecture: Shows multiple SMs along with their SPs. Device has global memory and Each SM has its own shared memory. SPs have local registers.

### 2.3 CUDA Programming model

*Compute unified device architecture (CUDA)* is a parallel computing architecture developed by NVidia. I list the following salient features.

- CUDA C provides a set of extensions to the ‘C’ programming language and as well as some low level and high level API.
- ‘C’ programmer can define CUDA kernels by writing ‘C’ style functions and CUDA kernels can be launched sequentially.
<table>
<thead>
<tr>
<th>Feature</th>
<th>GeForce 9400M G</th>
<th>Quadroplex 2200 S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Devices (GPUs)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>No. of SMs</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>No. of Cores per device</td>
<td>16 (2 x 8)</td>
<td>240 (30 x 8)</td>
</tr>
<tr>
<td>Clock speed per device</td>
<td>1.10 GHz</td>
<td>1.30 GHz</td>
</tr>
<tr>
<td>Global Memory per device</td>
<td>64MB</td>
<td>4GB</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64KB</td>
<td>64KB</td>
</tr>
<tr>
<td>Shared Memory per Block</td>
<td>16KB</td>
<td>16KB</td>
</tr>
<tr>
<td>Registers per block</td>
<td>8192</td>
<td>16384</td>
</tr>
</tbody>
</table>

Table 2.1: Sample NVidia GPU Hardware

- A CUDA kernel is executed as a *single program multiple data (SPMD)* on large data sets. Sample code fragment 2.1 shows a sample kernel code for addition of two 2D matrices.

- CUDA C programs are compiled using ‘nvcc’ compiler which generates ptx code. The *ptx* code contains a mix of *host* and *device* code. *Host* code is executed on the CPU and *device* code on GPU.

- Device code is executed with launch of CUDA Kernels (programs to be executed on device). The terms *kernel* and *program* are used interchangeably in the remaining of the document.

- Kernel launch configuration defines the number of threads to be executed. Configuration of *kernel* launch is defined by ‘block’ and ‘grid’.

- A *block* defines the number of threads per kernel and the *block* dimension can be 1D, 2D or 3D. A *grid* defines the number of *blocks* to be executed and its dimension can be either 1D or 2D. SMs of compute version 2.0 and above
support 3D grids. The hierarchy of the grids, blocks and threads are shown in the Figure 2.5.

- When a CUDA program on the host invokes a kernel, the blocks of the grid are enumerated and distributed among the available SMs. Threads in a block are executed concurrently and as a thread block terminates on an SM, new blocks are launched. Number of thread blocks launched in parallel depends on the number of SMs in the hardware.

- Concurrent launch of kernels is supported by some devices with compute capability 2.0 or above.

```c
// Kernel definition
__global__ void MatAdd(float* A, float* B, float* C, 
                         int width, int height)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[width*j+i] = A[width*j+i] + B[width*j+i];
}

... 

dim3 blockSize(4,4);

dim3 gridSize(1,1);

... 

int main()
{
...

    // Kernel invocation with 16 threads
    MatAdd<<<gridSize, blockSize>>>(A, B, C, 
                                     blockSize.x, blockSize.y);
}
```

Code 2.1: Sample Matrix Addition kernel
Figure 2.5: Grid & Block Structure: Kernel is launched with a grid of dimension 5 X 2. Each block in the grid is made up of 2D array threads. Thread block dimension for this sample is 5 X 3. Each of the 10 grids is launched one per *serial multiprocessor* and each *serial multiprocessor* executes the threads in a block concurrently.
CHAPTER 3

FMRI ANALYSIS AND CLUSTERING USING GPGPU

This chapter describes in detail the GPGPU implementations of various steps of the fMRI analysis pipeline and the GPGPU implementation of octree based hierarchical clustering. Section 3.1 gives an overview of the flow of analysis pipeline and clustering. In Section 3.2, I discuss parallelization of analysis and clustering algorithms.

3.1 Overview of the Analysis algorithm

In this section, I provide a brief overview of analysis methods presented in the work by Firdaus et al. [7]. The flow chart in Figure 3.1, shows the sequence of steps involved in the analysis pipeline and clustering. The stimulus function obtained from the experimental data is convolved with the hemodynamic response functions (HRF) to obtain the functional subspace. The HRF subspace obtained is then transformed with the Cohen-Daubechies-Feauveau 9/7 bi-orthogonal wavelet function (CDF 9/7) [12]. The lifting scheme of CDF 9/7 has been used which leads to a faster and in place calculation of the wavelet transform [16]. Similarly, the fMRI time courses are wavelet transformed using the same function CDF 9/7.
An orthogonal subspace of HRF basis wavelet is obtained by performing singular value decomposition (SVD) of the HRF basis. The wavelet transformed fMRI is projected on to the HRF subspace to obtain the feature vectors, which form the basis for the dissimilarity metric based clustering as explained in [7]. For more details on the statistical methods used, refer [7], [6], [13].

3.2 GPGPU Based Parallelization of fMRI Analysis

In this section, I shall discuss parallelization of functional elements shown in the flow chart. In the first section, I shall describe with the parallelization of modules in
the left branch of the flow chart of Figure 3.1 especially processing of experimental stimulus to obtain HRF subspace. In the second section I discuss the parallelization of the analysis of fMRI data including the pipeline module dealing with projection of wavelet transformed fMRI data on to HRF subspace.

3.2.1 Parallelization of Stimulus Function Analysis

Majority of the algorithms involved in stimulus function analysis except for singular value decomposition (SVD) are left to the CPU for processing, due to the very short execution times in the order of a few mill seconds. To achieve speed-up in the computation of SVD, CULA [1], a GPU accelerated version of LAPACK linear algebra library, has been used.

3.2.2 Parallelization of fMRI Volume Analysis

This section begins with a brief overview of the input data layout in the memory, followed by details about mapping between fMRI data and CUDA kernel threads. The remaining part of section deals with parallelization of analysis algorithms based on the mapping strategy discussed.

Data Layout Overview

The voxel to kernel thread mapping is based on the layout of the input fMRI volume in memory. Figure 3.2 shows the layout of fMRI data in memory. Input fMRI data is loaded into memory as an array of intensity values of each voxel for each of the volumes in the fMRI time course. The length of the array in memory is equal to size of each volume multiplied by length of the time course. Let, $W$: Be the width, $H$: Height and $D$: the depth. Let $N$: be the length of the time course.
Figure 3.2: fMRI data layout in memory as an array.

Length of the input array in this case is,

\[ L = W \times H \times D \times N \]

The total number of threads (total count of threads in all the blocks in the grid) that needs to be created for an analysis algorithm should be equal to number of voxels in a single volume in the time course i.e.,

Number of Kernel Threads = \( W \times H \times D \)

Each thread maps to a voxel at a given location \((x_t, y_t, z_t), \forall t \in T\) (‘T’: set of all time courses) in all the time courses. For example, Thread Id ‘0’ in block with Id ‘0’ is mapped to read the intensity values at voxel location \((0,0,0)\) in each of the volumes.
in the time course and store it in the thread local memory as an array and apply analysis algorithms on this array.

**Partitioning to blocks & grids**

fMRI image data can be divided into blocks and each block can be mapped to kernel thread block and each voxel in the fMRI image block is associated to a thread in the kernel thread block. This provides fine grained data parallelism and thread parallelism. Each sub problem can be solved independently in parallel.

CUDA C allows one, two or three-dimensional thread blocks. Thread ID of a thread at index ‘x’ in 1D block is ‘x’. In case of a 2D block with dimensions \((D_x, D_y)\), thread ID at index \((x, y)\) is \((D_x y + x)\). Similarly, thread ID for thread at index \((x, y, z)\), in a 3D block with dimensions \((D_x, D_y, D_z)\), is \((D_z D_y z + D_x y + x)\).

In this work, fMRI volume is treated as a 3D volume of vectors and each element in the volume is vector of length equal to length of time course \((T)\) and the intensities in each of the time course are treated as vector co-efficients. In essence, the input data is 3D volume of vectors and this 3D is divided into 2D sub-blocks, each of size \((D_x, D_y)\) and is mapped to equal sized blocks of threads, where each thread is mapped to one element (vector of intensities in time course) in the image data.

The sample code fragment 3.1 shows a sample code for dividing the 3D image into a grid of blocks using those values as in kernel launch configuration. In the kernel function, each element in the linear array containing the image data can be accessed using the thread IDs and block ID using a thread to voxel mapping strategy. The sample code fragment 3.2 shows a kernel call with thread ID to voxel mapping for indexing the elements of the image data.
// Volume extents
const cudaExtent volumeSize = make_cudaExtent(64, 64, 36);

// block size definition (32*16 threads per block)
dim3 blockSize(32, 16);
dim3 gridSize;
...
int main()
{
...
    // create grid of blocks based on volume extents
    gridSize = dim3(iDivUp(extent.width, blockSize.x) * \
                    extent.depth, iDivUp(extent.height, \
                    blockSize.y), 1);
...
    // Kernel launch
    KernelLaunch<<<gridSize, blockSize>>>();
}

Code 3.1: Sample division of blocks and grids

__global__ void KernelLaunch(VolumeTypePtr volPtr,\
cudaExtent extent)
{
...
    int x = blockDim.x * (blockIdx.x % iDivUp(\
                        extent.width, blockDim.x)) + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    int z = blockIdx.x / iDivUp(extent.width, blockDim.x);
...
}

Code 3.2: Sample kernel with thread ID to array mapping

The \((x, y, z)\) co-ordinate obtained using the thread ID to voxel location mapping strategy presented in code fragment 3.2 is used to index elements in the image data which is a linear array; the layout of which is present in the Figure 3.2. Intensity values are stored in rows first order. The order in which rows, slices and volumes are
Pitch calculations:

\[ \text{rowPitch} = W, \text{ Width of the Volume}, \]  
\[ \text{slicePitch} = \text{rowPitch} \times H, \text{ Height of the Volume}, \]  
\[ \text{volumePitch} = \text{slicePitch} \times D, \text{ Depth of the Volume}, \]

The \((x, y, z)\) co-ordinate obtained using mapping logic and pitch calculation equations 3.1 can used to index a voxel in the first volume of the fMRI time course using the equation:

\[ \text{index} = z \times \text{slicePitch} + y \times \text{rowPitch} + x; \]

The equation for accessing, voxel at \((x, y, z)\) in the \(K^{th}\) volume of the fMRI time series is:

\[ \text{index} = K \times \text{volumePitch} + z \times \text{slicePitch} + y \times \text{rowPitch} + x; \]

Using the equations (3.1), (3.2) and (3.3), voxels can be accessed from fMRI data and create the time course vectors. Code fragment 3.3 shows a sample code for reading the voxel intensities across the time courses and create intensity vectors. Each thread creates a vector for a voxel at a given location in each of the time series volumes and the vector is the input for data parallel algorithms, which are executed in parallel on the GPU.

The computed vectors 3.3 are used for determining the mean of the image, performing wavelet transform using CDF 9/7 and any other analysis calculations meant to be performed on each voxel along the time series independently.
```c
__global__ void SampleKernel(VolumeTypePtr vol, cudaExtent extent, int timeSeriesLength)
{

    // Calculation of \((x, y, z)\) Co-ords
    int x = blockDim.x *( blockIdx.x % iDivUpD(extent.width, blockDim.x)) + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    int z = blockIdx.x / iDivUpD(extent.width, blockDim.x);

    // To check if calculated co-ords are in the extent limits
    if(y < extent.height && x < extent.width &&
        z < extent.depth)
    {
        // Intensity vector at location \((x, y, z)\)
        float tSeq[length];
        size_t pitch = extent.width;
        size_t slicePitch = pitch * extent.height;
        size_t volPitch = slicePitch * extent.depth;

        // Voxel position in each volume
        unsigned int voxPos = z * slicePitch + y * pitch + x;

        // Iterating through the time course of data
        for(int i = 0; i < timeSeriesLength; i++)
        {
            tSeq[i] = vol[i * volPitch + voxPos];
        }
    }

    ...
    ...
}
```

Code 3.3: Sample code to create vectors of intensities for each voxel
3.3 Octree based Hierarchical Clustering of fMRI on GPU

In this section, I begin with a background on octree data structure, how they can be used for efficient spatial clustering, followed by previous work in literature. Later part of this section deals with the implementation details of octree based hierarchical clustering in the context of this thesis.

3.3.1 Background - Hierarchical Clustering

*Clustering* or *cluster analysis* is the task of creating conceptually meaningful groups of objects (called *clusters*) that share common characteristics, so that objects in same clusters are more similar to each other than those in other clusters. *Hierarchical clustering* (nested clustering) in statistics is a cluster analysis method used in building a hierarchy of clusters or a set of nested clusters that are organized as a tree [17]. Each node in the cluster tree (except leaf nodes) is a union of it’s child clusters.

*Clustering* approach can be either *top-down* or *bottom-up*. In a *bottom-up* or *agglomerative* approach, clusters are observed from the lowest level and pairs of clusters are merged while moving up in the hierarchy. Where as in a *top-down* or *divisive* approach, observations start at one cluster and divides recursively as it moves down the hierarchy.

Clustering of pairs of populations is determined by a dissimilarity metric and a linkage criteria. The *dissimilarity metric* or *distance metric* measures the distance (Euclidean distance, Mahalanobis distance [9], etc...) between pairs of populations. The *linkage criteria* determines the pairwise comparison between two distances and combines(*bottom-up approach*) or divides (*top-down approach*) the cluster. Some
commonly used linkage criterion are minimum or single-linkage clustering, maximum or complete-linkage clustering. In a single-linkage clustering, pairs of clusters are compared and the pair with minimum distance (based on distance metric) are clustered together, whereas in complete-linkage clustering the pair with maximum distance (based on distance metric) are clustered in the hierarchy.

3.3.2 Background - octree

An octree is a hierarchical tree data structure, in which each of the internal nodes have ‘exactly’ eight children and can be used to represent three-dimensional spaces. A three dimensional space can be recursively decomposed based on a octree, until it reaches the atomic element in space (a single voxel in this case). This has vast number of applications related to spatial clustering in three-dimensional space. Figure 3.3 shows sample octrees for 3D volumes of side lengths 1 (1 voxel) and 2 (8 voxels) and the possible number of levels of decomposition.

Figure 3.3: Sample octrees for 3D objects of side lengths 1 and 2 (length in terms of voxels)
3.3.3 Related Work

In the fMRI analysis, each voxels in the brain volume is compared with the every other voxel based on the dissimilarity metric [7]. Voxels exhibiting similar behavior are clustered together. The dissimilarity metric [7] penalizes the comparing voxels based on the their distance in space; the farther the distance, lesser the chances of clustering. Hence, clustering can be accelerated by only comparing and clustering each voxel with its immediate neighbors, which can be achieved using the octree hierarchical data structure. Further speedup gains can be obtained by parallelizing the construction of octree and clustering.

Clustering based on octree on a CPU is well understood. However, this is a compute intensive task primarily when it involves large 3D volumes. Parallelization of the octree based clustering can provide higher speedup gains. Such an implementation of construction of an octree on a GPU is addressed in [2], [4], Zhou et.al [19] have addressed the problem of surface reconstruction using data parallel octrees. However, hierarchical clustering based on an octree on a GPU is has not been addressed yet in literature. Also combined implementations (using both GPU and CPU) have not been addressed.

3.3.4 Implementation

Linearizing 3D volumetric data is very useful in the parallelization of octree construction. Particulary, linearizing along with grouping of the octant nodes as shown in Figure 3.5(b) in each octet of the octree together when dividing a node in the octree plays a significant role in division of data among the kernel threads.
Algorithm 1: Octree based Hierarchical Clustering (CPU version) [7]

**input**: Lower dimensional subspace of fMRI image data (3D volume with vector information)

**output**: Clusters at leaf level of octree

1. Decompose the brain volume into an octree.
2. Start at the given leaf nodes (Level 0) of the octree.
3. Create a cluster per voxel.
4. **foreach** parent (nodes at Level 1) **do**
   5. **while** until certain number of clusters are left **do**
      6. find a cluster pair \((C_i, C_j)\) and merge them to create a new cluster \(C_k \Leftarrow \text{Variance}(C_k = \{C_i, C_j\}) \forall i, j\)
      7. Remove clusters \(C_i\) and \(C_j\) from the set of clusters, and add \(C_k\)
   8. **end**
5. **end**
6. **if** \(\forall\) new clusters, in-cluster variance > threshold, \(K_L\) **then**
7. Union all the per-node cluster hierarchies
8. **end**
9. Move to the next level in the octree.
10. Repeat until the root node of the octree is reached.

*Space filling curves (SFC)* [14] are useful in linearizing 2D or higher dimensional data. Construction of octree data structure on a GPU using SFC indices of data has been addressed in [2]. The SFC Index value of cell in a 3D volume is computed by representing the integer coordinates of the cell using \(k\) bits for each dimension and by interleaving the bits. For example: For the coordinate \((1, 3, 5)\), with \(k = 3\), can be represented in binary as \((001, 011, 101)\) and the corresponding Z-SFC would be \(001010111\) \((87_{10})\). Figure 3.4 shows a sample computation of SFC Index from given 3D coordinate.

The internal layout of the voxel intensities for 3D volume over a time series has been described in section 3.2. Sorting of voxels (cells) based on their Z-SFC index, facilitates to group together the 8 octants of the octree. Figure 3.5(a), shows a sample
Figure 3.4: A sample computation of SFC Index from a given \((x,y,z)\) coordinate linear order of the 3D volume \(4X4X4\) before sorting and Figure 3.5(b) shows the cells after a non-decreasing ordered sorting based on Z-SFC Index and the grouping of the octants at the leaf level.

Code executing on GPU poses certain constraints which are not imposed while executing CPU code. One of the major constraints being no support for dynamic memory allocation. For this reason, a fixed amount of large enough memory has to be allocated for the new cluster array prior to the launch of the kernel. Similarly, array resizing is not allowed, which is required primarily while merging and removing of clusters. An efficient mechanism to overcome this constraint is illustrated in the Figures 3.7 and 3.8.

Figure 3.7 demonstrates the resizing of array while removing clusters with insignificant voxels at leaf level using shuffling mechanism. Figure 3.7(a) shows a set of clusters with a mix of both significant and insignificant voxel. Clusters with significant voxel are marked with a ‘1’ and all clusters with insignificant voxel are marked with ‘0’. All clusters with insignificant voxels are moved to the beginning of the array.
Figure 3.5: Linearizing and sorting of data based on Z-SFC Index. Sample volume of size 4X4X4 and $k = 2$.

and insignificant voxels to the end of the array. This is done by finding an cluster with insignificant voxel from the rear end of the array and replace it with cluster with significant voxel and index less than the insignificant voxel cluster. This loop is terminated when there are no more clusters with insignificant voxels and their indices greater than any cluster with significant voxel. The start index is updated with the index of the first cluster with significant voxel.
The clustering algorithm requires removal of clusters being merged and addition of newly created clusters which requires dynamic arrays or an array which is large enough to accommodate the existing clusters as well as the new clusters being created. For an input cluster array of size ‘n’, the maximum number of new set of merged output is \( n - 1 \). Hence the total array length required would be the sum of the total lengths of input array size and maximum possible length of the output clustered array i.e., \( 2n - 1 \).

Thread local memory is faster but a scarce resource which should be used judiciously. In the proposed mechanism, I exploit the fact that merged clusters can be removed and the same memory location can be used to store new clusters being created. At the end of each iteration of clustering, until only certain number of clusters are left, two clusters are removed and one cluster is added. Initially, two pointers \( StartIndex \) and \( EndIndex \) point to the beginning and the end of the array. The clustering starts by comparing the cluster at \( StartIndex \) with every other cluster. For the candidate clusters \( \{C_i, C_j\}; (i < j) \) and \( (i == StartIndex) \), identified to be merged, create a new cluster \( C_k \), replace \( C_j \) with \( C_k \) and increment the \( StartIndex \) by one step. This way in future iterations, the removed cluster \( C_i \) is not considered for clustering while \( C_j \) has been replaced with \( C_k \) and is considered in future iteration of clustering. In Figure 3.8, I demonstrate the use of constant and minimum size array in thread memory space for the purpose of clustering. The mechanism exploits the fact that a clustered child can be removed and this space can be used to store the created new parent cluster. All the intermediate clusters generated are stored in the intermediate cluster array (in global memory space).
Another constraint is in the requirement for use of pointers to maintain the parent-child relation while constructing the cluster hierarchy. The GPU can access and modify only the values in GPU memory address space. CPU cannot de-reference GPU memory address space and vice versa. Hence, the pointer references created for parent-child cluster relations during hierarchical clustering on the GPU, cannot be used by the Host (CPU) to traverse the cluster tree. To overcome this constraint, in the bottom-up approach of cluster construction, indices of child clusters in the child cluster are stored instead of pointers to them. The stored index can be used while dereferencing on the CPU.

The HRF coefficient vector values obtained by projecting wavelet transformed time courses on to HRF lower dimensional subspace are stored as 3D volume of vectors. This 3D volume of vector is used in the initialization of the leaf level clusters. Construction and initialization of leaf level clusters is detailed in the Algorithm 2. The cluster information is stored in the structure format as shown in the Figure 3.6. In the single linkage hierarchical clustering model, a parent cluster is constructed by merging exactly two child clusters, the corresponding indices in the child cluster array are stored in the nLeftChildIndex and nRightChildIndex respectively. Algorithm 3 details the per node (per octree parent node) hierarchical clustering. In each iteration corresponding to each level in the octree, number of GPU threads created is equal to number of nodes in the immediate parent level in the octree. For example, for a volume of dimension $8 \times 8 \times 8$ i.e., $8 \times 8 \times 8$ leaf level nodes in the octree, number of GPU threads required to cluster leaf level nodes is $8 \times 8$, and divide this by 8, as each time we move up each level in the octree.
Algorithm 2: Significant Voxel Filtering and Initialization of Clusters at Leaf Level of Octree

**input**: Lower dimensional subspace of fMRI image data (3D volume with vector information)

**output**: Clusters at leaf level of octree

// Construction of leaf level clusters
1 foreach dimensionSize 'd' in x, y, z of the volume do
2 if $d \neq$ nearest power of 2 then
3 Add padding data to adjust its side length equaling a nearest power of 2
4 end
5 end
6 Let the new dimensions be $x', y', z'$
7 Create an array of type cluster and of length $n = x' \times y' \times z'$
8 Create $n$ threads, with one thread per element in the array and do in parallel:

// Initialization
9 clusterPosition ← GetClusterPosition() // Refer 3.2
10 clusterVariance ← 0
11 leftChild ← −1 // ‘−1’ indicates no child found
12 rightChild ← −1
13 clusterVariance ← 0
14 SFCIndex ← GetSFCIndex(clusterPosition)
15 clusterFeatureVector ← lower dimensional subspace vector of the node

// Filtering of significant and insignificant voxels
16 if GetEnergy(clusterFeatureVector) < ENERGY_THRESHOLD then
17 isSignificant = TRUE
18 else
19 isSignificant = FALSE
20 end
21 Sort [5] the cluster array in non-decreasing order based on SFC Index
Algorithm 3: Agglomerative Hierarchical Clustering based on Octree Model

**input**: Clusters at leaf level of octree. Number of clusters, \( n = x' \times y' \times z' \)

**output**: Cluster Tree containing VOIs

// Construction of leaf level clusters
1. childArray ← Input Leaf Clusters
2. \( L ← n \)
3. for \( i ← \log_2 \max_{val \in x', y', z'} - 1 \) to 0 do
   
   Create an array, ‘parentArray’ of type cluster and of length
   \( L = (L \div 8) \times \text{CLUSTERS\_PER\_NODE} \)

   Create \( L \) threads, with one thread per octree node in the parentArray
   and do in parallel: // One thread per 8 octants in the child array
   
   parentIndx ← threadIdx.x
   childIndx ← threadIdx.x \times 8
   endIndx ← threadIdx.x \times \text{CLUSTERS\_PER\_NODE}
   localArray ← childArray [childIndx] to childArray [endIndx]
   
   foreach Cluster \( C_i \) in localArray do
      Remove \( C_i \) if \( C_i \) is at leaf level with single voxel and the voxel is not significant
   end

   while until \( \text{CLUSTERS\_PER\_NODE} \) clusters are left do
      
      for \( i ← 0 \) to Length0f (localArray) \(- 2 \) do
         for \( j ← i + 1 \) to Length0f (localArray) \(- 1 \) do
            \( C_i ← \text{localArray}_i \)
            \( C_j ← \text{localArray}_j \)
            if GetCombinedVariance (\( C_i, C_j \)) < \text{minVariance} then
               \( C_k = \text{Merge}(C_i, C_j) \)
               \( \text{minVariance} ← \text{GetCombinedVariance}(C_i, C_j) \)
               Remove \( C_i \) from localArray
               Remove \( C_j \) from localArray
               Add \( C_{ki} \) to localArray
            end
         end
      end
   end

   Copy the elements in localArray to parentArray starting at Index parentIndx
   
   Move one level up in the hierarchy considering parentArray as new set of child clusters

end
Cluster Struct
- int nNumOfVoxels: 0 // Total number of voxels in the cluster
- float3 clusterLocation: (0,0,0) // Location of the cluster
- int SFCIndx: -1 // SFC index of the cluster
- int nLeftChildIndx: -1 // stores index for left child cluster.
- int nRighChildIndx: -1 // stores index for right child cluster.
- float fClusterVariance: 0 // Combined variance
- float[] featureVector; // feature vector

Figure 3.6: Cluster structure format

(a) Before shuffling of voxels

(b) After shuffling

Figure 3.7: Shuffling of voxels in thread local array to group significant and insignificant voxels separately.

The generated cluster tree in the form of array of arrays, with each level of nodes stored in one array, a cluster tree is generated by going top-down i.e., starting with array ‘0’ which contains root level cluster and exploring its child clusters using the
rightChild and leftChild indices recursively. The tree generated is stored as an XML file.

3.4 User Interaction

The GUI based main window of the cluster explorer, a tool to explore clusters is shown in Figure 3.9 [15]. Cluster file (.xml) created by the application can be loaded and explored using this utility. The left view contains a tree control to navigate through the cluster hierarchy. For each cluster, the number of voxels and calculated mean and variance of the voxels can also be viewed. The right side of the view is for viewing the mean and ±1 standard deviation plots at each time point (standard deviation can be computed using equations in 3.4). The ‘mean’ envelope is in brown and the ±1 standard deviation plots are in black and green.

The standard deviation is given by:

\[
\{x_1, x_2, x_3, \ldots, x_n\} : \text{Set of ‘n’ voxels in a cluster} \quad (3.4a)
\]

\[
y_{x_i}(t) : \text{Measured time series} \quad (3.4b)
\]

\[
\text{Standard deviation, } \sigma(t) = \frac{1}{n} \sum_i [y_{x_i}(t)]^2 - \left[ \frac{1}{n} \sum_i [y_{x_i}(t)]^2 \right]^2 \quad (3.4c)
\]

The mean and the ±1 standard deviation plots can be used to observe the dispersion of the time series within the cluster by the user and can be used to identify a valid VOI region. The selected cluster (VOI region) can be saved as a raw image by selecting the ‘save Image’ item from the file menu of the Cluster Explorer. The saved cluster image can be overlaid on a structural image using tools like Mipav for viewing.
3.5 Results

In this section, I compare the performance of implementation of fMRI analysis and clustering algorithms on GPU with the corresponding implementations on CPU. The dimensions of the fMRI data used is $64 \times 64 \times 36$ and the length of the time course is 329. For performing wavelet transform, the length of the intensity vector is increased to 512 by padding with zeros. Also, for the ease of use in octree based clustering, the input data has been modified to have an extent of $64 \times 64 \times 64$ by padding with zeros. The new voxels added are treated as insignificant voxels while clustering.

Computing facilities at The Ohio Super Computing Center (OSC) have been used. The node used has a Quad-Core AMD Opteron Processor 2493MHz with 24GB of RAM and NVidia Quadroplex 2200 S4 GPUs with 4GB of global memory, 30 SMs and clock rates at 1.3GHz. The node has Red Hat Enterprise Linux Server 5.5 (Tikanga) OS installed on it. The installed CUDA driver version is 4.0 and CUDA Runtime is 4.0.

Optimizations: While clustering, optimal performance has been observed when the number of threads per block is in the range of 128-256 where each thread processes the child nodes of a single parent in the octree. To improve performance, use of global memory has been avoided and thread local memory has been used with the help of efficient mechanisms discussed previously (Section 3.7 and 3.8).

Figure 3.10 shows the CPU vs. GPU comparison of performance in calculating the mean of the image intensity field. Similarly, Figure 3.11, Figure 3.12 and Figure 3.13 show the performance comparisons for 1D computation of CDF wavelet transform, the computation of HRF coefficients by projecting wavelet transformed image on to the lower dimensional HRF subspace and filtering of significant voxels.
In order to realize the full potential of the GPU, the number of clusters on which hierarchical should be performed should be significantly large. This is evident from the Figure 3.14-3.18, which show the performance comparison results for each level in the octree with varying number of significant voxels. The number of clusters at the leaf level is significantly large and hence a significant performance difference is observed between CPU vs. GPU. CPU outperforms GPU at higher levels of the octree with less number of clusters (octree nodes), but at lower levels with higher number of clusters there is a significant achievement in performance by the GPU, due to greater realization of GPU potential.

Figures 3.19 and 3.20 shows the comparison of performance between CPU and GPU at Levels 6 and 5 of the octree. With the increase in number of leaf level clusters with significant voxels, the performance of CPU degrades whereas performance of GPU remains consistent. In Figure 3.20 for Level 5, the performance variation can be observed where the CPU is performing better with very small percentage of clusters with significant voxels and with increase in the number of significant leaf level clusters, GPU outperforms the CPU.

3.5.1 GPU+CPU Synergy

As explained earlier and from the results shown in Figure 3.14-3.18, it is evident that usage of GPU for clustering is beneficial only at lower levels of the octree which involves hierarchical clustering of large number of cluster groups. Based on the results obtained, in case of a very sparse to moderately sparse voxel scenario, the overall time taken for clustering (considering clustering at all levels in the octree), is much lesser in the case of using CPU alone ( Figure 3.21 ).
Based on these observations, it is evident that a combination of GPU and CPU can achieve even lesser overall clustering time. Clustering at lower levels is performed by the GPU. The obtained clustered data is passed to the host (CPU) for clustering at higher levels of the octree. The number of levels clustered by the GPU/CPU varies based on the sparsity of the voxels being clustered. The lesser the sparsity (higher number of significant voxels), the more the number of levels clustered by device (GPU). The higher the sparsity (lesser number of significant voxels), the more the number of levels clustered by the host (CPU). The combined use of the GPU and CPU achieves significant reduction in overall clustering time. Figure 3.21 shows a comparison of time taken in overall clustering for CPU, GPU and a combination of the GPU and CPU. Time taken for clustering using GPU and CPU combined (including time taken for data transfer between host and device), stays consistent with increasing number of significant voxels.

3.5.2 Proof of Applicability

As a proof of applicability, the generated hierarchical cluster tree which is stored as an XML file, is loaded in the Cluster explorer in Figure 3.9 for exploration of clusters and identification of valid VOIs. Clusters with significantly lower in-cluster variance and the ±1 standard deviation envelope vary by a small percentage when compared with the signal intensity envelope. Figure 3.23 vs. 3.24, shows the ±1 standard deviation and intensity plots for a VOI with poor quality and VOI with good quality, which is evident from the separation of ±1 standard deviation envelope from the intensity envelope. In Figure 3.23, the separation of ±1 standard deviation...
envelope and intensity envelope is higher compared to the 3.24. 3.25 shows cluster for the VOI identified in 3.24.
Figure 3.8: (a) shows ‘n’ (here, $n = 8$) child clusters belonging to single parent node. (b) Intermediate cluster array of size ‘$n - 1$’. (c) $C_2$, $C_4$ identified as candidate clusters to be merged. (d) Combined cluster $Ck_1$ is inserted in the intermediate array. (e) Modified child cluster array with the second cluster $C_4$ replaced with $Ck_1$ and modified start/end indices for next iteration. (f) No change in intermediate array. (g) $C_4$, $Ck_1$ identified as candidate clusters to be merged in second iteration. (h) Combined cluster $Ck_2$ is appended at the end of intermediate array. (i) Modified child cluster array with the second cluster $C_6$ replaced with $Ck_2$ and modified start and end indices for next iteration.
Figure 3.9: Cluster explorer GUI showing, cluster hierarchy in the left tree view control and the mean, ±1 standard deviation plots in the right side of the view. A cluster file (.xml) can be opened by selecting “Open cluster file” from “file” menu and the cluster image can be saved by selecting “save image” from “file” menu.
Figure 3.10: GPU vs. CPU: Calculation of Mean of the Image.

Figure 3.11: GPU vs. CPU: 1D Cohen-Daubechies-Feauveau 9/7 wavelet transform.
Figure 3.12: GPU vs. CPU: Computation of feature vector by projecting wavelet domain representation on to lower dimensional subspace.

Figure 3.13: GPU vs. CPU: Filtering of significant voxels based on the feature vector
Figure 3.14: GPU vs. CPU: Octree based clustering (19340 significant voxels)

Figure 3.15: GPU vs. CPU: Octree based clustering (27074 significant voxels)
Figure 3.16: GPU vs. CPU: Octree based clustering (32386 significant voxels)

Figure 3.17: GPU vs. CPU: Octree based clustering (36058 significant voxels)
Figure 3.18: GPU vs. CPU: Octree based clustering (38573 significant voxels)

Figure 3.19: GPU vs. CPU: Octree based clustering at Level 6
Figure 3.20: GPU vs. CPU: Octree based clustering at Level 5

Figure 3.21: Clustering: GPU and CPU combined for varying number of significant voxels.
Figure 3.22: Clustering: GPU and CPU combined considering all voxels as significant.

Figure 3.23: ±1 Standard deviation and intensity plot for a poor quality VOI
Figure 3.24: ±1 Standard deviation and intensity plot for a good quality VOI

Figure 3.25: Valid cluster identified using cluster explorer
fMRI analysis holds a significant role in detecting the activations in brain and temporal relationships between different brain functions for a given task the subject is performing during the scan. This analysis is in general very compute intensive. Hierarchical clustering of populations is another compute intensive task, primarily when it involves large populations like large volume size in case of an fMRI.

With the emergence of graphics cards containing GPUs allowing GPGPU programming and further rapid developments in the performance of the GPUs opened the doors for new form of heterogeneous computing. Their ability to execute data parallel algorithms at faster rates compared to a traditional CPU, attracted the attention of developers to solve several compute intensive problems in several domains. In this thesis, I exploit the capabilities of GPU to accelerate the execution of fMRI analysis and hierarchical clustering based on an octree. I present parallelized pipeline including the calculation of the mean of image intensities, conversion of fMRI time series to wavelet domain, computation of the feature vectors, filtering of insignificant voxels and most important being the clustering based on the octree data structure.

I compared the parallelized versions with their CPU equivalent counterparts and showed that the GPU versions out perform the CPU versions for all the algorithms.
except for clustering. In case of clustering, the CPU version outperforms the GPU for smaller populations but in case of large populations, the GPU is realized to its full potential and outperforms CPU significantly. Since the number of clusters is very small at higher levels of octree, the use of CPU is ideal rather than GPU for clustering. However, since the overall clustering time taken by GPU is higher than CPU for sparse voxel scenario, I proposed a method for using a combination of GPU and CPU together to achieve speedup in overall clustering.

I applied the developed GPU algorithms to analyze the fMRI data of subjects and the generate an output cluster tree. Also, I presented an GUI based application to explore the clusters and determine correct VOIs. The user is aided by the graphical display of the mean and ±1 standard deviation plots while identifying the VOIs.

These algorithms can be used to solve a wide variety of problems which are compute intensive and involve 3D/4D volumetric data and also problems involving agglomerative clustering.

With the use of combination of GPU and CPU, I achieve significant speed-up gains compared to using GPU or CPU alone. Further gains in speed can be achieved by using the optimization techniques listed below:

1. **Loop Unrolling**: Flow control instructions like *if, while, for etc.*, significantly impact the performance. Unrolling the loops can contribute in achieving higher speed-ups. Similarly, the performance can be improved by eliminating some of the *if conditional statements*.

2. **Sparse voxel clustering**: Number of significant voxels that are considered while clustering is considerably very less when compared to the total number of voxels. Ignoring insignificant voxels during the construction of leaf level clusters can
improve the performance of hierarchical clustering at leaf level. This eliminates some of the conditional flow control instructions.

3. Using GPU and CPU: In this thesis, I discussed the use of a combination of GPU and CPU to achieve performance, by clustering the lower levels of the octree using GPU and higher levels using CPU. Currently, the number of levels to be clustered by GPU/CPU is manually tuned. A mechanism can be incorporated to allow the system to make an intelligent decision that switches between GPU and CPU while clustering.
BIBLIOGRAPHY


[10] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. NVIDIA Corporation, 06 2011. 9


