Parallelizing LINQ Program for GPGPU

by

Pritesh Agrawal

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

Parallelizing LINQ Program for GPGPU

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

Master of Technology (M.Tech)

by

Pritesh Agrawal

Supervised By

Dr. Amey Karkare

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

POST GRADUATE OFFICE
THESIS SUBMITTED
ON 20·06·2012
I. I. T., KANPUR - 16

CERTIFICATE

It is certified that the work contained in this thesis entitled
" *Parallelizing LINQ Program for GPGPU*",
by *Pritesh Agrawal(Roll No. 10111028)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 19/6/2012

(Dr. Amey Karkare)

Department of Computer Science and Engineering,

Indian Institute of Technology Kanpur

Kanpur-208016

June, 2012

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor **Dr. Amey Karkare** for his support and encouragement. He provided me freedom to experiment with new ideas and inculcated the techniques required to understand and progress my work. I am grateful for his patient guidance and advice in giving a proper direction to my efforts. I would also like to thank the faculty and staff of the Department of CSE for the beautiful academic environment they have created here.

I would like to thank all my friends of M.Tech CSE 2010 Batch for making these past two years a memorable one for me. I especially thank **Adarsh, Ajith, Kunjan, Sumit, Yash** and colleagues of room **CS305** for their company and long discussions.

Last, but not the least, I would like to thank my parents, my younger sister and my friend **Pragya** who have been a constant source of support and encouragement. Their belief in me survived even as mine failed.

Pritesh Agrawal

Dedicated to

my parents and my sister.

ABSTRACT

Recent technologies have brought parallel infrastructure to general users. Nowadays parallel infrastructure is available in PC's and personal laptops. Now single core machines have became history. Even multi-core technologies are replaced by GPGPUs when it comes to high performance computing because GPGPUs are giving many cores at low cost. Sequential programs of the past are unable to efficiently utilize this parallel architecture. An application which run parallel has lesser running time than sequential application. Writing parallel programs manually is a difficult task. So we can not expect domain experts to write parallel programs. We can create an automatic parallelizing compiler to convert a sequential code to parallel code. Domain experts can use this kind of compiler to convert their sequential code to parallel code to utilize parallel infrastructure.

In this work, we have proposed a tool which will convert a sequential code to parallel code. Here for sequential code we took LINQ programming language and for parallel architecture we took CUDA. LINQ is a query language developed by Microsoft to query data in .NET Languages and CUDA is an architecture developed by NVIDIA to use GPUs. Our proposed parallelizing compiler will automatically convert a LINQ code to an equivalent CUDA code. Microsoft has also developed a compiler to parallelize LINQ operators but it is only for multi-cores and not for GPGPUs. In our work we are parallelizing these LINQ operators in GPGPUs using CUDA.

Contents

List of Figures

List of Tables

List of Programs

Chapter 1

Introduction

Some real life applications and simulations like Physics-based simulations, Aerodynamic and streamlining simulation, Digital image processing, Video Processing, Bio-informatics, Database operations which deals with huge amount of data, requires huge amount of resources and takes lots of execution time. It is important that such programs make efficient use of resources for better running time performance.

Modern computers are not getting faster than previous generation. Instead we have to add more and more CPU's to perform multiple tasks in parallel that is called *multi-core processors machine.* As the number of CPU's are increasing it is becoming more complex to write code as it has to handle interaction of large number of threads.

To run those applications which require heavily parallel computing, one of the well know technique is *data parallelism.* In data parallel computation same computation is performed on different data sets. That means we can execute same code on different cores or threads on different data. Some well known data parallel programming environments are High Performance Fortran(HPF) [HPF97], NVIDIA's Compute Unified Device Architecture (CUDA) API for graphics processor [NVI11] and Google map/reduce framework [MR04]. All these architecture require very complex programming as we have to interact with threads and processors.

The aim of this thesis is to describe an environment that let user program in sequential manner and produce automatically equivalent data parallel code that will run on GPGPU [GPG12].

1.1 GPGPU

In each generation of CPU, improvement of computational power is the main area of research. After continuously improvement CPU's performance, the growth is not enough to compare it with GPU. This difference in performance has drawn the interest of researchers to look for GPUs as a possible solution. This high computational requirements has led to the emergence of new term General-purpose computing on graphics processing units(GPGPU).

Here we will present some key points to compare CPU and GPGPU.

1.1.1 High Computational Power

If we talk about computational power of GPUs and CPUs then GPU's computational power is much more than CPUs computational power. This difference in computational power is because of design model of CPUs and GPUs. CPU is basically a group of few cores with lots of cache memory that can handle few software threads at a time, but GPU is group of hundreds of cores that can handle thousands of threads at a time. Computational power of GPU can be 10 to 100 times better than CPU because of hundreds of cores to process thousands of threads, which depends on degree of parallelization of program.

1.1.2 Memory Bandwidth

Memory bandwidth is a term to represent rate of data transfer between processor and memory. In other words rate of data fetch from memory or data store to memory is called memory bandwidth. Memory bandwidth is a very important issue when comparing CPU and GPU. Memory bandwidth of GPU is generally higher than CPU. This is very important for GPU as it should be able to fetch data fast enough to meet the data need for its large number of cores and threads. Memory bandwidth between CPU and Main memory is in order 1∼2 Gbps whereas in GPU it is in range 75∼150 GBPS.

1.1.3 Programming Model

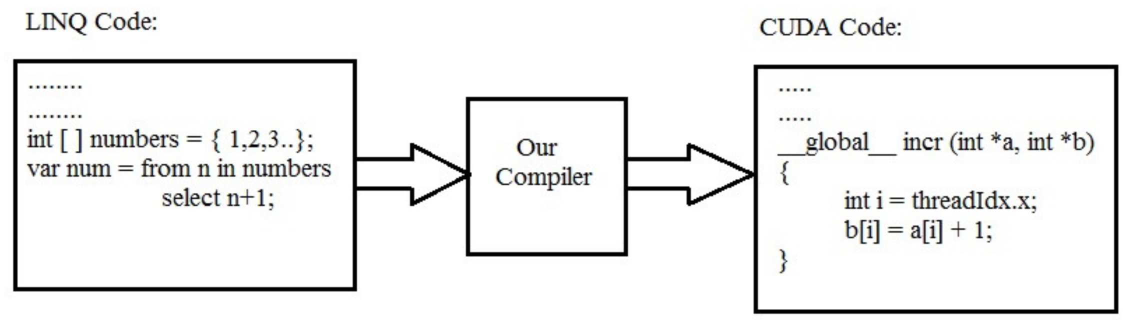
GPGPU is a generic term used for framework which allows non graphics applications to run on graphic processors. Various such framework are present like, CUDA [NVI11], OpenCL [Ope12], Stream SDK for ATI GPUs. All these platforms are popular while programming with GPGPU.

CUDA is one of the famous framework out of those. A large number of applications in different fields are running in CUDA, but still number of tools available to code for sequential languages are more than for any GPGPU framework. Any GPGPU framework requires that code should be written in data parallel format but it is not necessary for CPU code. It is a tough task to write a code in data parallel format than writing in sequential format. Despite this difficulty GPGPU programming model is becoming popular with time due to very high computing power of GPU which requires in many fields of engineering and scientific applications.

1.2 Motivation

Automatic conversion of sequential code to parallel code has been a topic of research for several decade. Various tricks and techniques are used by compiler community to achieve parallelization such as parallel design models. Many of libraries were used to do this task like Message Passing Interface(MPI), Pthreads, OpenMP. Theses tools and libraries made the task easier. If we have multi cores architecture in machine then this approach is very useful. We can get better run time by this method. But if we have installed GPU in our machine then are we utilizing our resource properly? The answer to this question is no, we are only using multi cores not GPUs.

In this work we are presenting data parallelization using GPGPUs, for this we



```
........
........
int [ ] numbers = { 1,2,3..};
var num = from n in numbers
            select n+1;
```

```
.....
.....
__global__ incr (int *a, int *b)
{
        int i = threadIdx.x;
        b[i] = a[i] + 1;
}
```

Figure 1.1: Convert LINQ to CUDA.

need one language which deals with huge amount of data and we found query language is a good option for this. So we started our work with query language and then we parallelize it. For this task we choose Language integrated query (LINQ) [LIN12b] as query language which deals with huge amount of data. And then later we converted LINQ code to GPGPU code. For GPGPU we used Compute Unified Device Architecture (CUDA).

1.3 Our Work and Contribution

Our work is aimed to domain specialize programmers, who knows their problems best like physicist, aerospace engineers, database specialist, but they are not experts in writing parallel codes. They know there are lots of opportunity for parallelization in their problem but they don't know how to use tools and libraries of parallel programming. Basically, their work is a very long running simulation and calculation, where a little bit of parallelization can reduce their running time in order of hours, days or even months. In this work we parallelize LINQ codes in GPGPU by using CUDA libraries so that we can save lots of time. Detailed discussion on LINQ is in section 2.1.

As we know in many simulation we need to execute our code on huge amount of data. We know query language is a better option when dealing with large set of data. LINQ is one of the best choice as one of the query languages, because LINQ codes interact with objects as well as with data. LINQ makes a query a first-class language construct in C# and Visual Basic.

We started our work for a real life algorithm and first implemented it in C# then replaced loops by LINQ operators and after that we converted LINQ code to GPGPU code for CUDA library. Fig 1.1 will help you to get a clear picture of our work. In this processes we made some observation while converting a LINQ code to GPGPU code and applied those observation to code each LINQ operator in CUDA. Later those observation will be useful when writing a converter for LINQ to CUDA.

1.4 Organization of Thesis

In **chapter 2**, we explained background knowledge which we required for our work. In which we explained about LINQ and LINQ operators, Similarities between LINQ and Haskell and then a brief knowledge of execution of LINQ queries. After that we explained about CUDA in details.

In **chapter 3**, we introduced data parallelism. Then we took a real life algorithm as our example and implemented it first in C# then replaced loops by LINQ operators and finally we implemented it in CUDA. So basically this chapter represent our implementation from starting to end for an example of our choice.

In **chapter 4**, we explained CUDA code for each LINQ operator. And later we discussed issues which we faced in our work.

In **chapter 5**, we showed running time in CPU as well as in GPU of our CPU version of code and GPU version of code.

In **chapter 6**, we explained some researches which are related to our work and also we will see how they are different from our work.

In **chapter 7**, we explained the conclusion of our thesis and also we discussed future work which can be performed based on this thesis.

Chapter 2

Background

This chapter describes the concepts which are prerequisites to understand our work. We will start our discussion with LINQ, and explain how LINQ is different from other programming languages. After that we will describe about CUDA library used for GPGPU programming.

2.1 Introduction to LINQ

We present an overview in this section of LINQ operators, Query writing and query execution. Here we will only discuss in brief which is relevant to our work.

LINQ was introduced by Microsoft in .NET Framework version 3.5 in 2007 and runs in Visual studio 2008. LINQ reduce the gap between world of objects and world of data. The syntax and semantics of LINQ are influenced by SQL and Haskell. LINQ can be used with .NET Framework collections, SQL Server databases, XML documents, and ADO.NET Datasets.

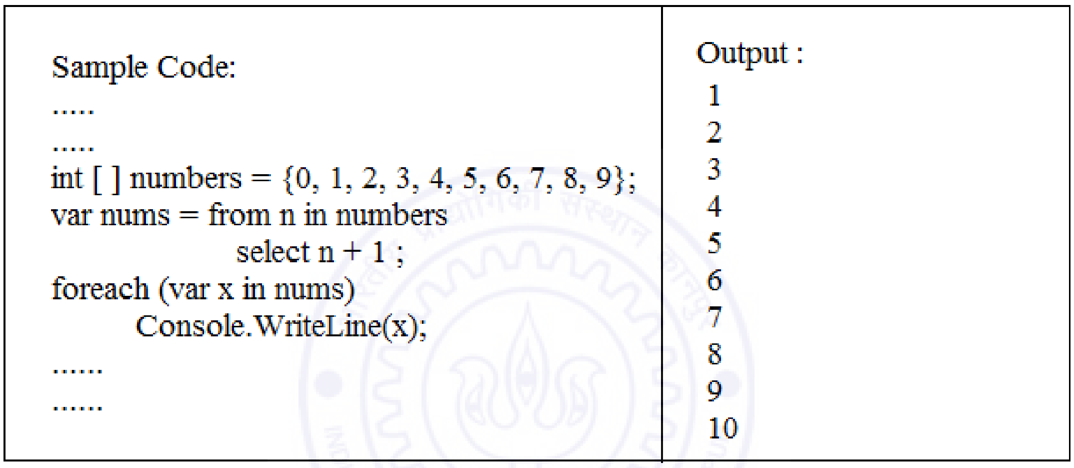
A group of functions are included in LINQ which are known as standard query operators along with translation rule from query expression to expressions using lambda calculus. These operators are used to insert, modify and delete data from databases languages like SQL and functional languages like Haskell.

2.1.1 Standard Query Operators

The query operators defined by LINQ are known to the user as Standard Query Operators. Those operators are as following:

i. **Projection Operator**

Projection operator for LINQ is *'select'*. It is used to project (apply) some function in selected data elements from the list. For sample code see fig 2.1.



```
Sample Code:
.....
.....
int [ ] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
var nums = from n in numbers
                select n + 1 ;
foreach (var x in nums)
        Console.WriteLine(x);
......
......
```

```
Output :
1
2
3
4
5
6
7
8
9
10
```

Figure 2.1: Sample code for Projection Operators.

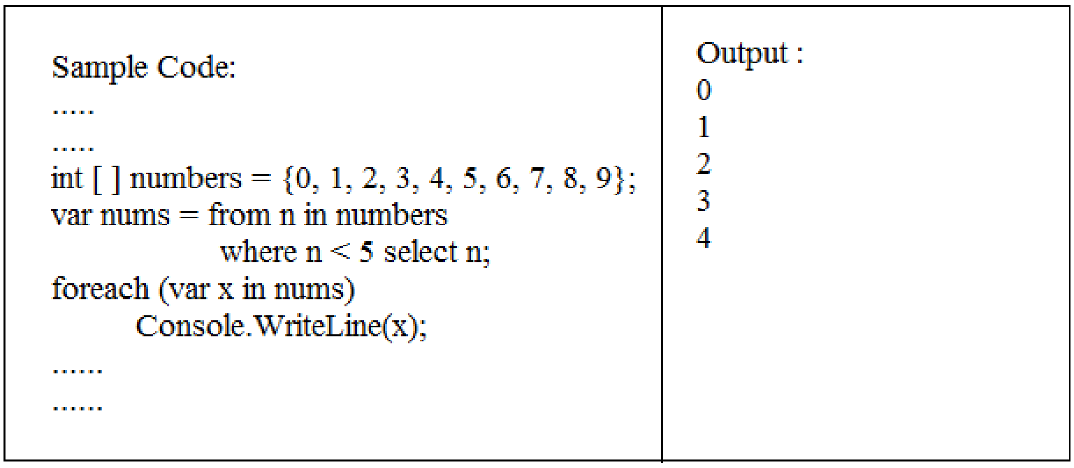
ii. **Restriction Operator**

Restriction operator for LINQ is *'where'*. It is used to restrict some elements from a list as given in condition. For sample code see fig 2.2.

iii. **Partitioning Operators**

Partitioning operators are to create partition of list. It also partitions list for a given condition. For sample code see fig 2.3.

Operators : *Take(), TakeWhile(), Skip(), SkipWhile().*



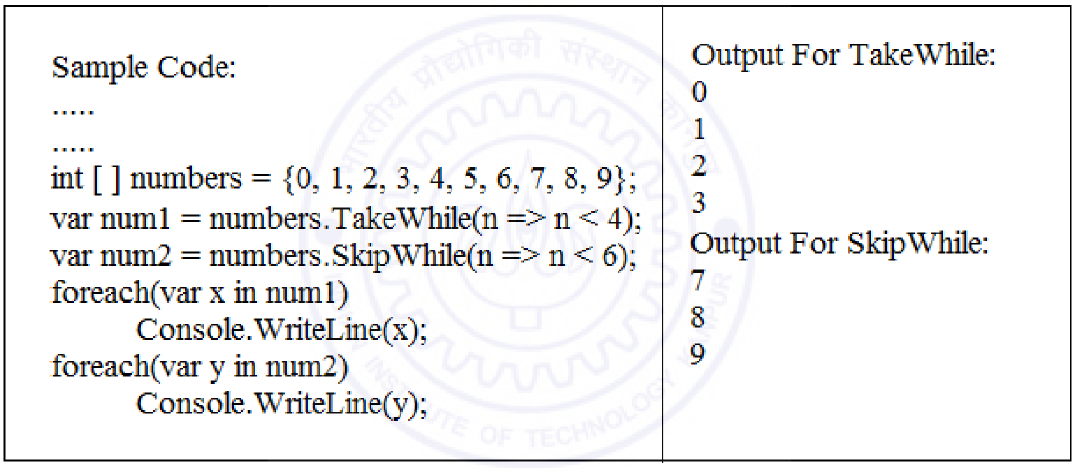
Sample Code:

```
.....
.....
int [ ] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
var nums = from n in numbers
                    where n < 5 select n;
foreach (var x in nums)
          Console.WriteLine(x);
......
......
```

Output :

```
0
1
2
3
4
```

Figure 2.2: Sample code for Restriction Operators.



Sample Code:

```
.....
.....
int [ ] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
var num1 = numbers.TakeWhile(n => n < 4);
var num2 = numbers.SkipWhile(n => n < 6);
foreach(var x in num1)
          Console.WriteLine(x);
foreach(var y in num2)
          Console.WriteLine(y);
```

Output For TakeWhile:

```
0
1
2
3
```

Output For SkipWhile:

```
7
8
9
```

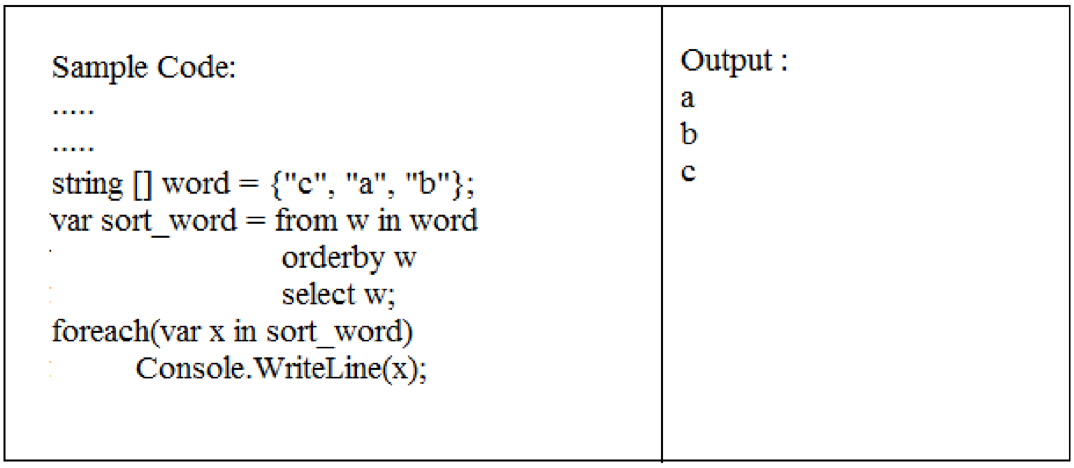
Figure 2.3: Sample code for Partitioning Operators.

iv. Ordering Operators

This operator is used to order list. The operator is *'OrderBy'*. For sample code see fig 2.4.

v. Grouping Operators

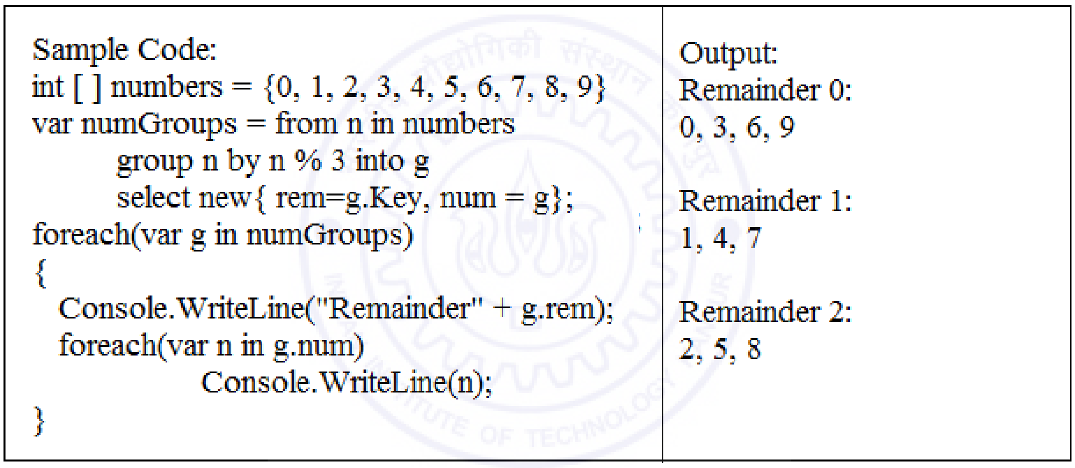
This operator used to form group for list. Operator is *'Group var By'*. For sample code see fig 2.5.



```
Sample Code:
.....
.....
string [] word = {"c", "a", "b"};
var sort_word = from w in word
                  orderby w
                  select w;
foreach(var x in sort_word)
        Console.WriteLine(x);
```

```
Output :
a
b
c
```

Figure 2.4: Sample code for Ordering Operators.



```
Sample Code:
int [ ] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
var numGroups = from n in numbers
        group n by n % 3 into g
        select new{ rem=g.Key, num = g};
foreach(var g in numGroups)
{
  Console.WriteLine("Remainder" + g.rem);
  foreach(var n in g.num)
              Console.WriteLine(n);
}
```

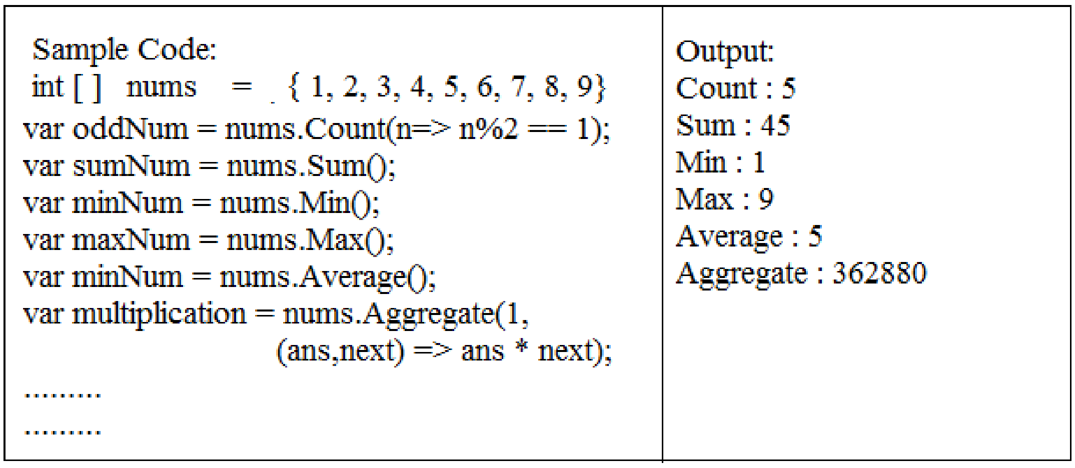
```
Output:
Remainder 0:
0, 3, 6, 9

Remainder 1:
1, 4, 7

Remainder 2:
2, 5, 8
```

Figure 2.5: Sample code for Grouping Operators.

vi. **Aggregate Operators**

This operator is used to give a result as single value for a list. For this operator takes a *lambda* that specifies what operation has to be done to combine two elements of a list and it continues the result till the end of list. For sample code see fig 2.6.

Operators : *Count(), Sum(), Min(), Max(), Average(), Aggregate().*



```
Sample Code:
int [ ]  nums  =  { 1, 2, 3, 4, 5, 6, 7, 8, 9}
var oddNum = nums.Count(n=> n%2 == 1);
var sumNum = nums.Sum();
var minNum = nums.Min();
var maxNum = nums.Max();
var minNum = nums.Average();
var multiplication = nums.Aggregate(1,
                        (ans,next) => ans * next);
.........
.........
```

```
Output:
Count : 5
Sum : 45
Min : 1
Max : 9
Average : 5
Aggregate : 362880
```

Figure 2.6: Sample code for Aggregate Operators.

2.1.2 Similarity Between LINQ and Haskell

As mentioned earlier LINQ is influenced by Haskell therefore both the languages share a lot common among themselves. Haskell [Has12] is a purely functional programming language which supports higher order functions. A high order function is a type of function which can take a function as an argument or can return a function as a result.

Higher order functions example:

```
def f(x):
    return x + x
def g(function, x):
    return function(x) * function(x)
print g(f, 5)
```

*Ans: (g (f , 5) = (5 + 5) * (5 + 5)) = 100*

Similarity between Haskell and LINQ is given in table 2.1

LINQ: **Standard Query Operators**	**Haskell:** **Higher Order Function**
Restriction Operators *Where*	*filter*
Projection Operators *Select*	*map*
Partitioning Operators *Take, Takewhile, Skip, SkipWhile*	*takeWhile, dropWhile*
Ordering Operators *OrderBy*	*OrdereBy*
Grouping Operators *Group By*	*Group By*
Aggregate Operators *Count, Sum, Min, Max, Average, Aggregate*	*foldl, foldr*

Table 2.1: Similarity Between LINQ: Standard Query Operators and Haskell: Higher Order Function

2.1.3 LINQ Queries

Queries are used to get data from database. Queries are represented by query languages. Various query languages has been developed to access data from different type of data source, like SQL for relational databases and XQuery for XML. So the developer has to learn a new query language every time he encounters a new kind of data source. LINQ provides solution to this problem by giving a consistent model for programming with query language for different types of data sources. In LINQ we are supposed to perform three steps to run a program:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

Working of above written steps is given in an example below. In this example we have taken an integer array as data source for our convenience. We can use the same logic for other data sources. In LINQ, query creation and query execution are two different steps. Only by creating query variable we wont be getting any data from the data source until we wont execute query.

```
class LINQcode
{
   static void Main()
   {
        // The Three Parts of a LINQ Query:

         //  1. Data source.
           // Basically here we are using integer array as a data
           // source for data source logic is same.

        int[ ] numbers = new { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

         // 2. Query creation.
           // numQuery is an IEnumerable<int>
           // this will only create query we are not getting any
              data
                // after execution of this line.
        var numQuery =
                from num in numbers
                where (num % 2) == 0
                select num;

         // 3. Query execution.
           //After execution of this line we will get data
           // here 'num' will contain data each time we will
           //get one item from data source
        foreach (int num in numQuery)
        {
                Console.WriteLine(num);
        }
   }
}
```

Program 2.1: Example: To copy data from one array to another array in multiprocessor environment with two processors.

2.2 Introduction to CUDA

Here in this section we will discuss about CUDA. To program and execute kernel in NVIDIA GPUs we need to use CUDA libraries. By using CUDA libraries, GPU are represented as highly multi threaded architecture to the CPU, and it is used to execute parallel programs.

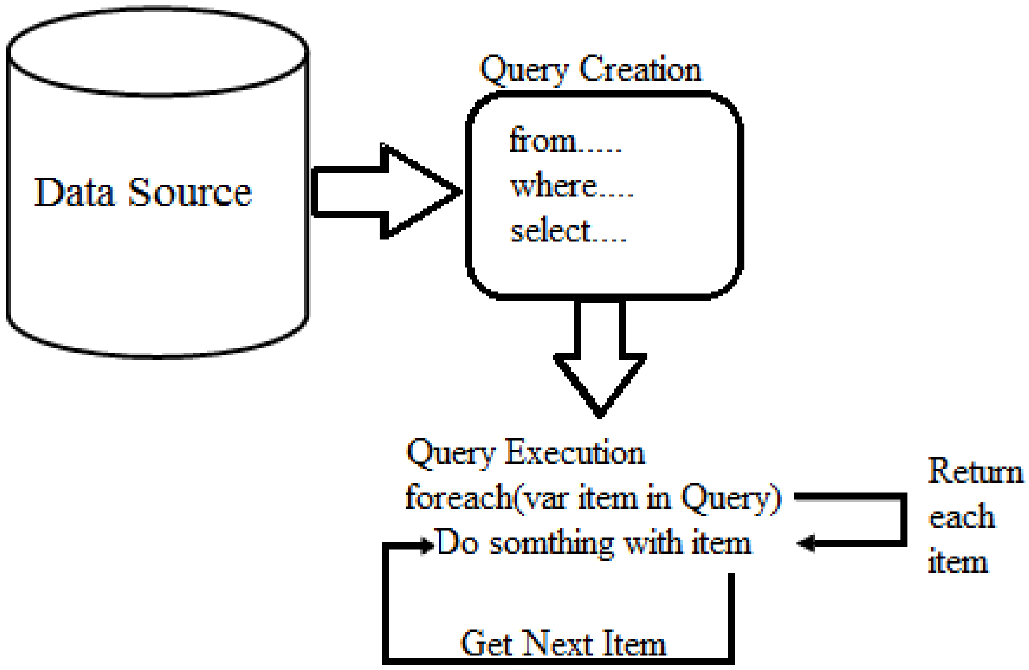


Figure 2.7: LINQ programming model with three part of query operation.

2.2.1 Execution Model

As we know CUDA executes program in multi-thread and multi-processor, which gives it highly parallel environment. In fig 2.8 parallel environment of CUDA and control flow of execution from CPU to GPU is also shown. When CPU calls a device function or GPU function then control flow goes from CPU to GPU.

GPU architecture is divided into three levels grids, blocks and threads. Thread-processors run threads, multiprocessors run blocks, and device runs grids. Each grid contains many blocks either in one dimension and in two dimensions. Each block contains lot of threads in one dimension, two dimensions and in three dimensions. Maximum number of blocks in one grid should not exceed *65,535* and maximum number of threads in one block should not exceed *512*. Fig 2.8 contains a two dimensional architecture of CUDA.

By looking architecture of CUDA it is clear that it runs their code in Single Instruction Multiple Threads(SIMT) fashion. SIMT is very useful in data parallelism,

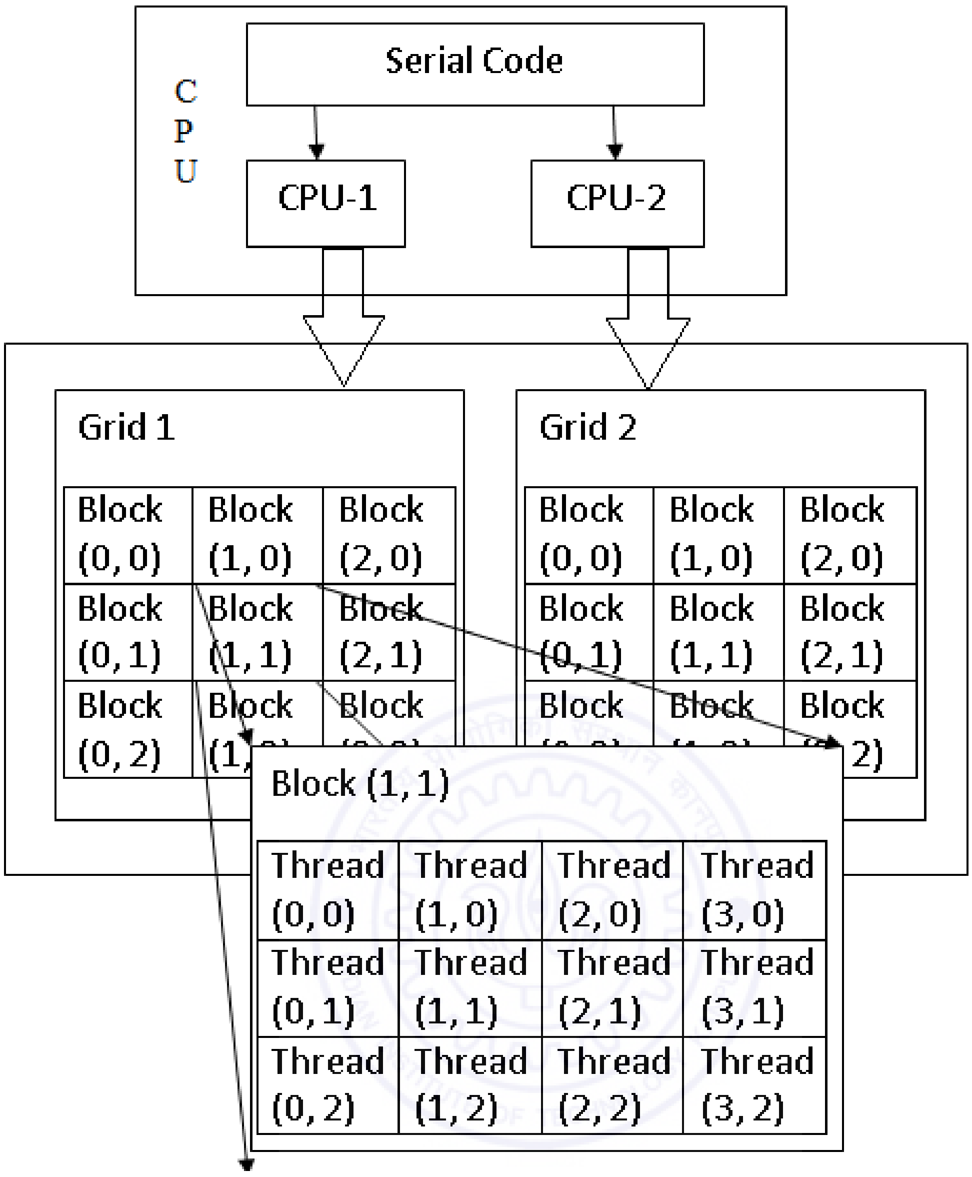


Figure 2.8: Code execution model in CPU and CUDA. [NVI11]

as instruction is same for all data set which runs in different threads for different set of data.

2.2.2 Programming Model

To use GPU as a general purpose we need to code in C/C++ programming language by using NVIDIA CUDA library. A GPU code contains two parts, first part is run in CPU which is called host program, another part contains set of device functions which is used to invoke CUDA kernel, is called device code. that are generated from the host code to GPU or device. Device functions are called from host to move control from CPU to GPU. These functions start with a keyword *__global__*.

Set of threads and blocks are created by invoking device kernel, these threads and blocks are accessed by keyword *threadIdx* and *blockIdx.* keyword.
For one dimension threads a thread index with *'x'* will have threadID x.
For two dimensional threads a thread index with (x,y) will have threadID $(x+y*D_x)$ if block size is (D_x,D_y).
For a three dimension threads a thread index with (x,y,z) will have
threadID $(x+y*D_x+z*D_x*D_y)$ if block size is (D_x,D_y,D_z).

For one dimensional architecture:

```
__global__ ArrayAdd(int A[N], int B[N], int C[N])
{
        int i = threadIdx.x;
        C[i] = A[i] +B[i];
}
```

For two dimensional architecture:

```
__global__ MatAdd(int A[N][N], int B[N][N], int C[N][N], )
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] +B[i][j];
}
```

2.2.3 Memory Model

CUDA Memory model is distributed in multilevel memory as shown in fig 2.9. For each thread CUDA has a local memory called register which is the fastest memory available in CUDA memory. Register of one thread is not accessible by the another

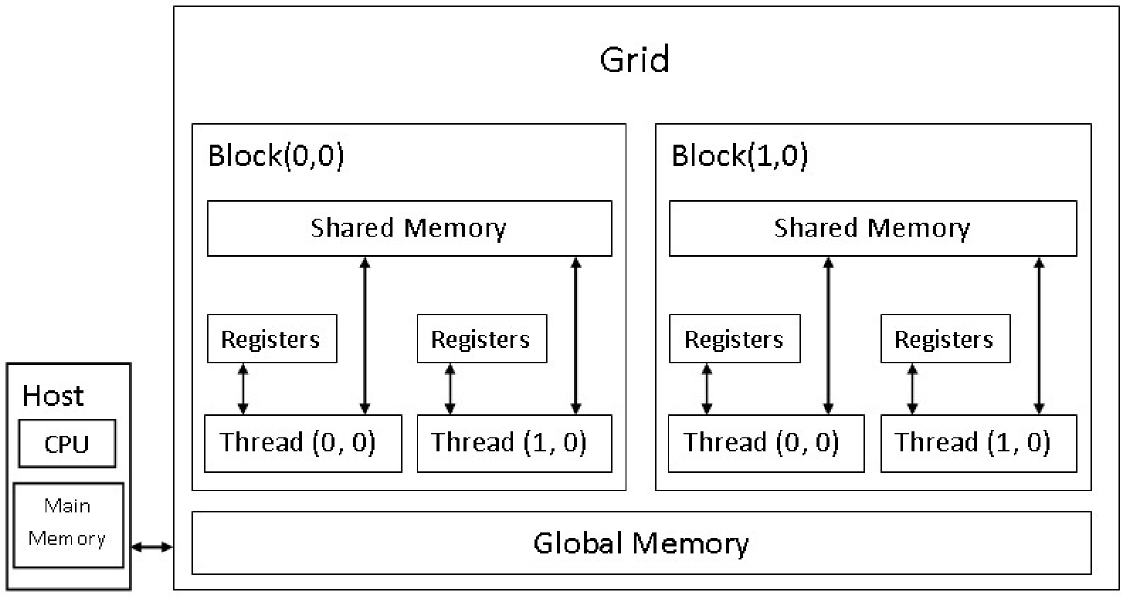


Figure 2.9: CUDA Memory Model. [NVI11]

threads even if they belong from same block. Next level of memory is called shared memory, for each thread block there is a separate shared memory. All threads of a block can access shared memory of their block but threads from different blocks can not access it. Next level of memory is called global memory, which is accessible by all the threads from all the blocks. Bandwidth of global memory is very high.

To allocate memory in any device we use *cudamalloc()* function. This function creates memory in global memory space in device which is accessible by all the threads of all the blocks. To allocate memory for threads within a block or to allocate memory in shared memory space we need to declare it using prefix *__shared__*.

2.2.4 Thread Scheduling Mechanism

In CUDA, whenever a kernel function is executed it actually executes one grid in device. Each grid is a set of parallel blocks and threads. Whenever all threads complete their execution, the grid gets terminated. After that control goes to host until next kernel is invoked. To utilize full hardware of GPU we need to create large number of threads. Applications with high data parallelism gives better results on GPU.

Each grid is divided into blocks and then each block is divided into threads. All blocks must contain equal number of threads organized in the same manner. To run blocks there are *Streaming Processors(SMs)*. Each block in grid is executed in one SM through out its lifetime. One SM can not run two blocks parallel. If number of blocks are more than number of SMs in device then extra blocks are queued for SMs in uniform fashion. If two blocks are running in same SM that means SM will finish execution of first block and then start execution of second block, this is more likely that two blocks are running sequentially without any parallelization.

After completion of blocks to SMs mapping, each block is divided into group of threads in the next level. This group of threads is called WARPs. A group of 32 threads create one WARP. this is a standard size but it also depends on implementation. Scheduling in GPU is done according to WARP. Threads from same block are arranged in WARP in a sequential manner like thread ID 0 to thread ID 31 will be in first WARP, thread ID 32 to thread ID 63 will be in second WARP and so on. At a time one SM can execute only one WARP, but here WARP of the same block can be scheduled in same SM. If one WARP is busy with IO work then other WARP which is in ready state can be scheduled in that SM. If there are more than one WARP in ready condition then one of them is selected on some priority basis.

Chapter 3

An Overview of Our Work

In this chapter we will describe data parallel in LINQ using CUDA. For this we have selected one real time application rather than series of small examples, namely the Barnes-Hut algorithm for N-body simulation [BH86]. In this application there is lot of opportunities for parallelization that's why this application could be a good example to show our work. We started our work with coding in C# then we converted all loops by LINQ operators and then we wrote equivalent CUDA code. By this we got a parallel LINQ code in CUDA for this algorithm. Here we will discuss coding in details.

3.1 Data Parallelism

Parallelization of code enables code to run simultaneously on different computing units or processors. Main advantage of parallelization is on execution of a large program because large codes are divided into many small codes and then it run on all processors simultaneously. Data parallelism is a method of parallelization of code in which data is divided into small groups and then execution is done on each group simultaneously. In data parallelism generally code is same for all data so that it is easy to run it parallel for different groups of data. Data parallelism is generally used when there is huge amount of data. To implement data parallelism lots of threads are created and executed on different processors simultaneously.

```
void DoThisTask(int a[], int b[], int N)
{
        if (CPU = 0)    //For CPU number 0
        {
                min = 0;
                max = N/2;
        }
        else            //For CPU number 1
        {
                min = N/2;
                max = N;
        }
        for(i = min; i < max; i++)
                b[i] = a[i]+1;
}
```

Program 3.1: Example: Copy data parallelly from one array to another array in multiprocessor environment with two processors.

3.2 *N*-Body Barnus-Hut Simulation Algorithm

Now we will discuss algorithm of application which we used in our work i.e. N-Body Barnes-Hut simulation algorithm [BH86]. We will describe this algorithm in details, will make some observations and in chapter 4 we generalize those observation so that they are applicable for parallelization of any sequential program.

In our work we will restrict this algorithm for two dimensions to get more clarity. We will not discuss three dimensional examples here and we will also avoid the complications because of bodies that are very near to each other.

An n-body simulation algorithm is used to calculate the motion of 'n' bodies or 'n' particles under the gravitational force. In brute force solution we calculate force between each pair of particles which takes O(n^2) time to compute result.

The Barnes-Hut algorithm used to reduce time complexity to O(nlogn) by grouping some particles which are close enough to each other and using those particles as a single body, that single body is represented as *center of mass* of all the particles of group.

The center of mass of a group of particles is the average of positions of particles in that group, weighted by mass. If there are two particles, location of first particle is (x_1, y_1) and location of second particle is (x_2, y_2) and mass of first particle is m_1 and mass of second particle is m_2, then there total mass m and center of mass (x,y) will be:

```
m = m1 + m2
x = (x1*m1 + x2*m2) / m
y = (y1*m1 + y2*m2) / m
```

Force exerted by the group of particles on a body is approximated by the force exerted by center of mass of those particles, if those particles are sufficiently far away from body. The word sufficiently far away depends on the accuracy of the final result which can be maintained accordingly.

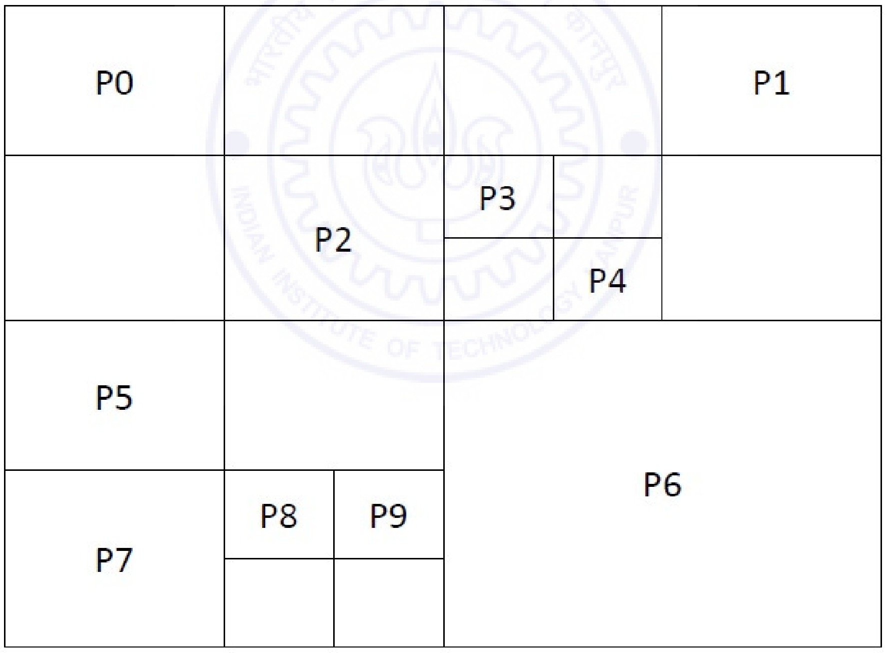


Figure 3.1: Example: Particles in Area and division of Area.

In Barnes-Hut algorithm the particles are divided into groups according to their location and stored in a quad tree data structure. A quad tree contains at most 4 children for each node. Each node represents an area. In other words the area is split into four regions of equal size and each region is represented by one node of quad tree. After that particles are divided into four groups according to their location. Each group of particles is collection of all particles for one sub area. Then center of mass is calculated for each group. We recursively repeat this step for each sub group of particles until a group has only one particle or zero particle.

Figure 3.1 shows the working of this algorithm for some particles. Let p_0, $p_1 ....p_9$ are the particles for which we have to run this algorithm.Initially we divide area into 4 sub areas. Then we divide all particles in four groups according to their location. For the given particles lower right quadrant has only one particle so this quadrant will not be divided further. In upper left quadrant there are two particles so we need to divide it more. But after one more division both particles are in different quadrant so we need to stop here for this quadrant. For upper right and lower left quadrant there are more particles and after one more iteration some of quadrant have more than one particles so we need to divide these quadrant again till all the particles are in a single quadrant.

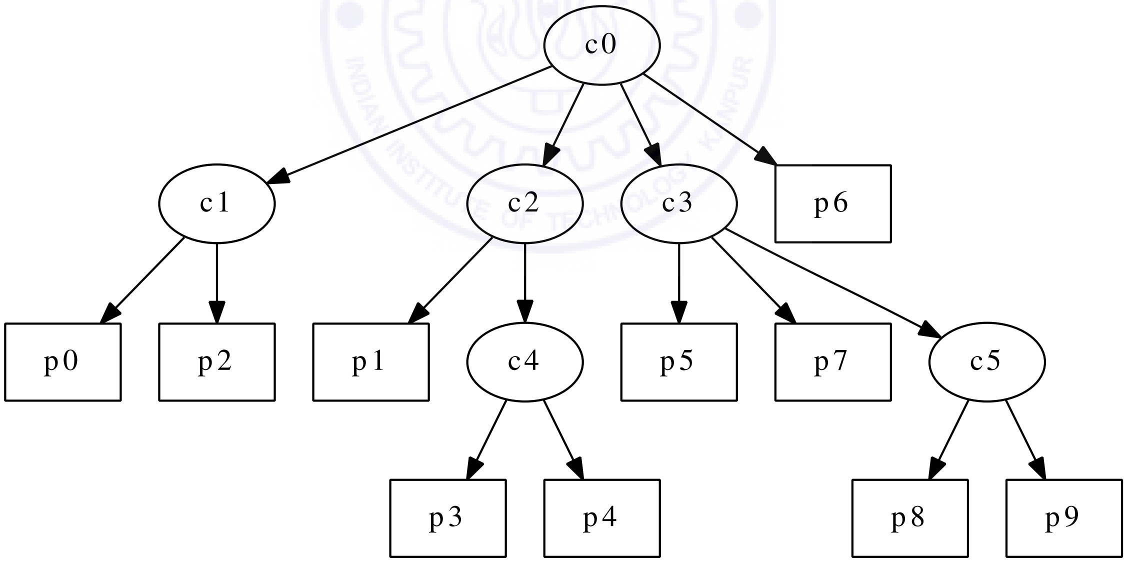


Figure 3.2: Example: Quadtree representation of fig 3.1.

Figure 3.2 shows the quadtree structure of particles p_0, $p_1 ....p_9$. Root node c_0 is the center of mass of all the particles. Root node of each subtree is center of mass of all the particles of that subtree or all the particles which are inside sub-area represented by that root node. If it is a single node then that is particle itself.

Creating quadtree and calculating center of mass for each sub-area is first phase of Barnes-Hut algorithm.

In second phase of algorithm forces on each particle are calculated. This is done by traversing the tree from the root to downwards. Let p be a particle for which we have to calculate force, then for each subtree from root if the center of mass is sufficiently far away from particle p then calculate force using center of mass and do not look down in that subtree and if center of mass is not sufficiently far away then do this step recursively for each subtree of that center of mass.

3.3 Encoding Barnes-Hut Algorithm in C# and LINQ

As mentioned earlier we are implementing Barnes-Hut algorithm in C# by using LINQ operators and then will convert it in CUDA. The goal of our work is to identify the opportunities of parallelism in a sequential LINQ program. therefore we developed our program in 3 steps. The first step is to code Barnes-Hut algorithm in C# without using any LINQ operator. Then in next step we will use LINQ operators to replace simple C# loops, and in last step we will write equivalent code in CUDA. Now we will see each step in detail.

3.3.1 Coding in C#

Recall that Barnes-Hut is used to find out forces exerted by particles on other particles. The input of each particle contains three values mass, location and velocity. We used data type *Particles* to represent a particle in code. Data type *Particles* is using a data type *Location* to store the location of particle in (x,y) format. Fig 3.2 is segment of code to show all data types which is used to implement Barnes-Hut algorithm.

```
Location
{
        double x;       // x co-ordinate of Loaction.
        double y;       // y co-ordinate of Loaction.
}

Particles
{
        Location loc;   //Data Type Location:
                            //(x,y) co-ordinates.
        double m;       //mass of Particles.
        double vel;     //velocity of Particles
        double force;   //Forces exerted on Particles
}
Area    //Square shape area
{
        Location min;   // Left-Below Point
        Location max;   // Right-Upper Point
}

TreeNode        //Node of QuadTree
{
        List<TreeNode> child;   //To store subtree
        Particle centerofmass;  //Centerofmass of subtree
        bool Ispoint;           // Is contains point or not?
}
```

Program 3.2: Data type used in implementation of Barnes-Hut algorithm.

Area and Division of Area

Area data type is used to represent area in which particles are present. Initially all the particles are present in *InitialArea*. The root of quadtree represents the region *InitialArea*. Data type *Area* has two data members of type *location* which contains left-below point and right-above point of an area. As initial area is a square and in each iteration we are dividing it into 4 squares that means it will be always a square after any iteration, so we can represent this square by two points. In each iteration area is divided into four parts, as shown in fig 3.3.

Class Area contains a function *SplitArea()* that is used to split area into 4 parts. Function *SplitArea()* returns an array of type *Area* with size four which contains 4 sub-areas of input area.

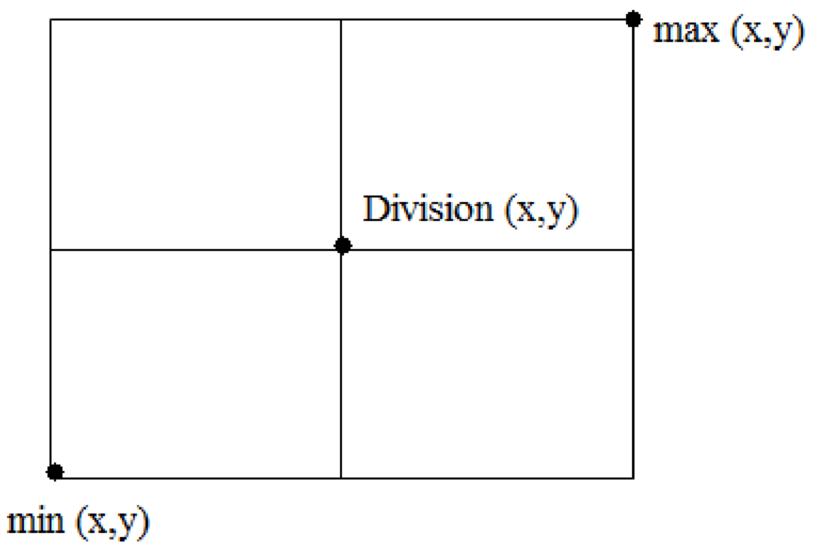


Figure 3.3: Area Data type with two data memebers and division of area.

```
    public Area[] SplitArea(Area input)
    {
        Area[] subarea = new Area[4];
        Particles division = new Particles();
        division.x = (min.x + max.x)/2;
        division.y = (min.y + max.y)/2;

        subarea[0].min.x = min.x;
        subarea[0].min.y = division.y;
        subarea[0].max.x = division.x;
        subarea[0].max.y = max.y;

        subarea[1].min.x = division.x;
        subarea[1].min.y = division.y;
        subarea[1].max.x = max.x;
        subarea[1].max.y = max.y;

        subarea[2].min.x = min.x;
        subarea[2].min.y = min.y;
        subarea[2].max.x = division.x;
        subarea[2].max.y = division.y;

        subarea[3].min.x = division.x;
        subarea[3].min.y = min.y;
        subarea[3].max.x = max.x;
        subarea[3].max.y = division.y;

        return subarea;
    }
```

Program 3.3: Function *SplitArea*.

This class also contains a function *inArea()* that used to tell a given particle is inside a given area or not. This function takes *Particle* as input and returns a boolean value; if *Particle* is inside the given area then it returns *True* else it returns *False*.

```
public bool inArea(Particles point)
{
  return ((point.x >= min.x)&&(point.x<max.x)
        &&(point.y >= min.y)&&(point.y<max.y));
}
```

Program 3.4: Function *InArea*.

Creation of Quadtree

As discussed in section 3.2, for all particles we need to create a quadtree according to their locations by using function *SplitArea()* and *inArea()*. To create quadtree we use a data type *TreeNode* that is used as a node of tree details of data type *TreeNode* is given in figure 3.2. In our code we used function *CreateTree()* to make quadtree. *CreateTree()* takes two arguments as input, one is area and other is a list of particles. The input area is the region that is represented by current node of tree and the list of particles contains all input particles that belong to this area. Now we have to calculate center of mass of all the particles. Calculation of center of mass is shown in program 3.5.

```
  //'input' is List of all the particles
   for (int i = 0; i < LengthofInput; i++)
       mass += input[i].m;
   centerofmass.m = mass;
   for (int i = 0; i < LengthofInput; i++)
   {
       centerofmass.x += (input[i].x * input[i].m);
       centerofmass.y += (input[i].y * input[i].m);
   }
   centerofmass.x /= mass;
   centerofmass.y /= mass;
```

Program 3.5: Calculation of Center of Mass.

After calculating center of mass we need to divide inputs particles in four part according to there location by using four subareas which we will get by using *SplitArea()* function.

```
//'NextInput' is a list of list where each list will
//    contain list of particles for four subareas:
 subarea = SplitArea(inputarea);
 for(i=0; i<4; i++)
 {
    for(j=0; j<LengthofInput; j++)
    {
       if (subarea[i].InArea(input[j]))
          List[i] = input[j];
    }
 }
```

Program 3.6: Dividing particles in four lists.

After getting four different lists of particles for four different areas we will call same *CreateTree()* function recursively for each list. The recursion terminates when the list has a single element or is empty.

```
void CreateTree(Particles[] input, Area inputarea)
{
   .... // code for centerofmass
   if(input.Size == 1)
   {
      ispoiunt = True;
      return;
   }
   .... // code to divide area
   .... // code to create 4 lists of particles
   for(i=0; i<4; i++)
   {
      if(list[i].Size >0)
         child[i].CreateTree(list[i], subare[i]);
   }

}
```

Program 3.7: Function call CreateTree recursively.

Force Calculation on Particles

To calculate force on a particle we traverse quadtree from top to bottom. In each step we find if center of mass is sufficiently far away from particle then we calculate force exerted by center of mass else we call same function recursively for their children untill we reach a leaf node. After reaching a leaf node we calculate force exerted by leaf node. Here we need to use a threshold distance which will decide approximation of result. In this function we use a variable *theta* which decides particle is sufficiently far away from center of mass or not.

```

public double ForceCal(Particles onpoint, double theta)
{
   if (Ispoint) /* Current Node is a point or not. */
   {
      if (!((point.x==onpoint.x)&&(point.y==onpoint.y)))
         return centerofmass.m;
      else
         return 0; /* This is the point itself */
   }
   double d, force=0;;
   d=Math.Sqrt((onpoint.x-centerofmass.x)*
                  (onpoint.x-centerofmass.x)+
                (onpoint.y-centerofmass.y)*
                  (onpoint.y-centerofmass.y));
   if (length / d > theta)
   {
      for(i=0;i<4;i++)
         force += child[i].ForceCal(Particles onpoint,
                                          double theta);
      return force;
   else
      return centerofmass.m;
}
```

Program 3.8: Force Calculation.

3.3.2 Coding in LINQ

As discussed in section 2.1 LINQ queries are used to manipulate data from data set by using LINQ operators. LINQ operators execute query in each data element of data set one by one. LINQ operators work on arrays or lists, so while writing LINQ code we need to use mainly arrays or lists.

In C# code which discussed in previous section having most of the data type in form of arrays or lists. Here in LINQ code we replaced loops of previous code by LINQ operators. As we used most of the data type in arrays or lists form so we did not need to change definition of data type. In this section we basically focused on loops and replaced those loops by LINQ operators. Now we will discuss the changes that we made.

Area and Division of Area

In class *Area* there are two functions *SplitArea()* and *inArea()*, for these two function we don't need to use any loops as in function *inArea()* we are comparing two points and in function *SplitArea()*we are dividing given area in four sub-area. So we did not change anything in this part.

Creation of QuadTree

At time of creating quadtree we need to calculate center of mass of all the particles. To calculate center of mass we used loops in simple C# coding. Now here in this part we replaced this loop by LINQ operators. While calculating center of mass we operated some function in all the elements of list one by one and got one value as a result. In other words we can say we reduced given array to one element. LINQ operator *Aggregate* is used to reduce an array to one element. Modified code after replacing loops by *Aggregate* operator is shown below in Program 3.9.

```
    //'input' is List of all the particles
    centerofmass.m = input.Aggregate(0.0,(ans,next)
                      => ans + next.m);

    centerofmass.x = input.Aggregate(0.0,(ans,next)
                      => ans + (next.m * next.x));

    centerofmass.y = input.Aggregate(0.0,(ans,next)
                      => ans + (next.m * next.y));

    centerofmass.x /= centerofmass.m;
    centerofmass.y /= centerofmass.m;
```

Program 3.9: Calculation of Center of Mass using LINQ operators.

After calculating center of mass next step is to divide input particles in four lists according to their location in 4 sub-areas, Where sub-areas can get by using function *SplitArea()* by dividing given area into four parts. Here we are using three different arrays; first is to store four sub-areas, second is to store four lists of particles, and third is to store four children nodes of current node. We need to use LINQ operator on these three arrays at same time, that will be very complex. To reduce complexity we took a wrapper class which will contain all these three data types. Size is same for those three arrays so it is easy to use wrapper class. Program 3.10 shows the definition of wrapper class. We create array of objects of this wrapper class and apply LINQ operator on this array.

```
class wrapper
  {
    Area divide;
    List<Particles> InputNext;
    TreeNode NextChild;
  }
```

Program 3.10: Wrapper Class to contain three object in one place for LINQ operators.

Now we need to divide input particles in four lists according to their location in area. For this we divided area int four sub-areas by using function *SplitArea()* and then by using function *inArea()* we divided input particles in four lists. Here we are restricting some particles in final result by looking their return values of function call *inArea()*. To restrict some elements we used restriction operator *'where'*of LINQ. Code for dividing input particles by using *'where'* operator is given below in Program 3.11.

```
//'Next' is array of object of class wrapper.

 for(int i=0; i<4; i++)
 {
    var item = from current in input where
                Next[i].divide.inArea(current)
                select current;

    foreach(var N in item)
       Next[i].InputNext.Add(N);
 }
```

Program 3.11: Dividing particles in four lists using LINQ operators.

We have four different lists of input particles along with four sub-areas respectively. To create subtree for four child previously we used loop and made tree for each child recursively, but now we replaced this loop by LINQ operator. This loop is basically projecting *CreateTree()* function into four children so we can use projection operator in place of loop. We know *'select'* is projection operator so we used *'select'* operator in our code.

Modified code to create tree by using *'select'* operator is in Pragram 3.12.

```
void CreateTree(Particles[] input, Area inputarea)
{
   .... // code for centerofmass
   if(input.Size == 1)
   {
      Ispoiunt = True;
      return;
   }
   .... // code to divide area
   .... // code to create 4 lists of particles

   var NextChild = from variable in Next select
                     variable.NextChild.createTree
                     (variable.InputNext,variable.divide);

   int j=0;
   foreach (var ChildN in NextChild)
   {
      child[j] = ChildN;
      j++;
   }
}
```

Program 3.12: Function call CreateTree recursively by using LINQ operators.

Force Calculation on Particles

As discussed above to calculate force on particles we need to traverse whole quadtree from top to bottom. Previously in simple C# coding we used loops but here we will replace loop by LINQ operator. In force calculation we are adding four forces that is result of four subtrees, so *Aggregate* operator can do our work easily.

Modified version of force calculation by using *Aggregate* operator is shown in Program 3.13.

```
public double ForceCal(Particles onpoint, double theta)
{
   if (Ispoint)
   {
      if (!((point.x==onpoint.x)&&(point.y==onpoint.y)))
         return point.m;
   }
   double d;
   d = Math.Sqrt((onpoint.x-centerofmass.x)*
                  (onpoint.x-centerofmass.x)+
                (onpoint.y-centerofmass.y)*
                  (onpoint.y-centerofmass.y));
   if (length / d > theta)
      return child.Aggregate(0.0,(current, next)
          =>current+next.ForceCal(onpoint, theta));

   return centerofmass.m;
     }
```

Program 3.13: Force Calculation using LINQ operators.

3.4 Barnes-Hut Algorithm in CUDA

As discussed above, we are converting a LINQ code into CUDA code, for which initially we described structure of LINQ code. Now we will explain coding of Barnes-Hut algorithm in CUDA. As we know in CUDA we have lot of threads and processors, to use those threads and processor effectively we need to divide our code into those threads and processors. In this algorithm we have lot of opportunities to divide code into threads and processors. The structure we used to parallelize our code is described in details below.

3.4.1 Parallelization in Tree Creation

In section 3.3.2 we discussed tree creation in LINQ. We know each node of tree contains four children equivalent to four subareas. At the time of tree creation in function *CreateTree()* for all children code is same and they are totally independent to each other. So basic idea for parallelization is in first level create four threads for each child of root and run code of one child in one thread as shown in fig 3.4.

Till now we used only four threads to run this code parallel but we know in

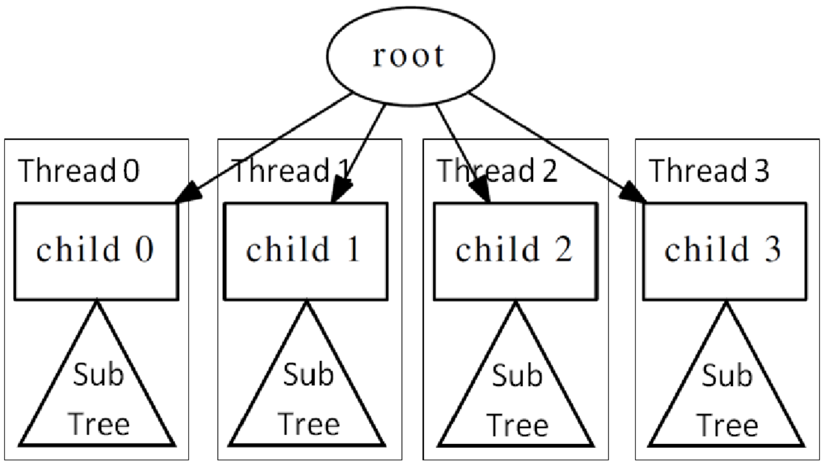


Figure 3.4: Running code on 4 Threads and one block.

CUDA we have lot of threads and processors. To use resources more effectively we need to parallelize this code with more threads. To use more threads in this code we modified this method for 16 threads. In quad tree root has 4 children and each child of root has also 4 children so in 2^{nd} level there are 16 children. Now we can run each subtree rooted on these 16 children on 1 thread. By this method we are using 16 threads and running on 16 threads instead of 4 threads. Graphically representation of this method is in fig 3.5.

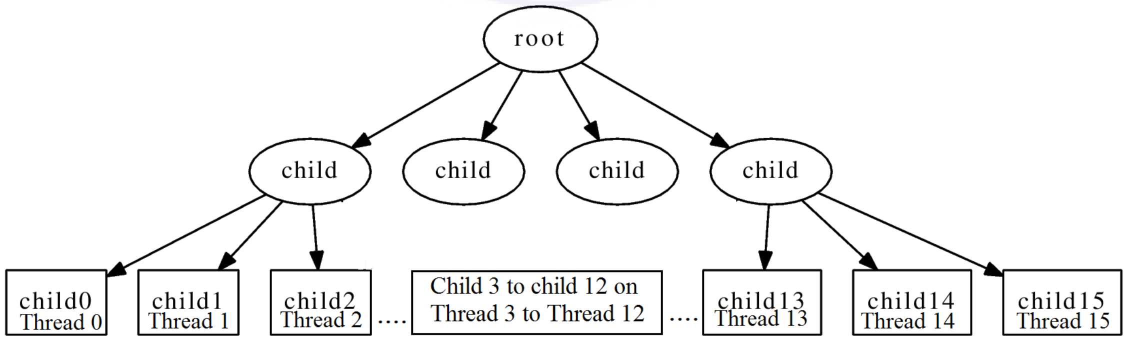


Figure 3.5: Running code on 16 Threads and 1 block.

In method discussed above we are using 16 threads but we are using only one block of GPU. For more efficient use of GPU we need to use more blocks. Now we will describe our next modification in code for better utilization of resources by dividing code into more blocks. In this modification we took 16 children at depth level 2 and run each child into one block, by this we used 16 blocks. Now for each subtree rooted at those 16 children we took another 16 children at depth level 4 and run those 16 children in 16 threads. Total we run 16 blocks and each block runs 16 threads that means we run total 16 blocks and 256 threads as shown in fig 3.6.

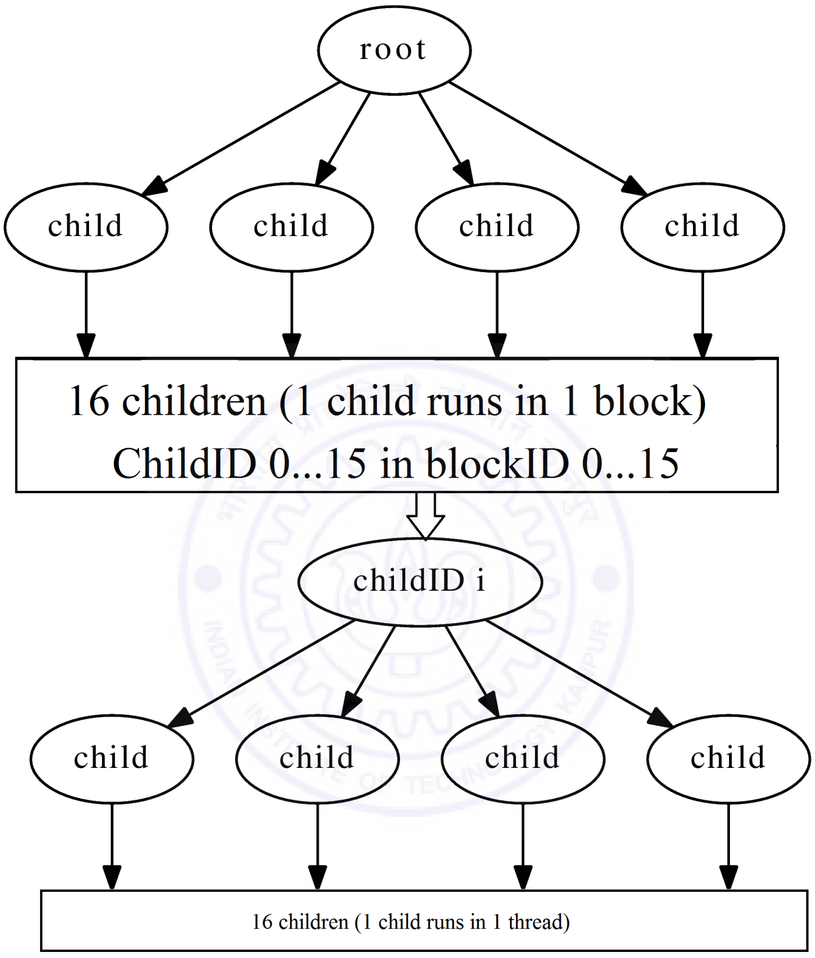

root
child
child
child
child
16 children (1 child runs in 1 block)
ChildID 0...15 in blockID 0...15
childID i
child
child
child
child
16 children (1 child runs in 1 thread)

Figure 3.6: Running code on 256 Threads and 16 blokcs.

By running code on 256 threads and 16 blocks we used GPU resources effectively and parallelize most of the part of this code for tree creation.

3.4.2 Parallelization of Force Calculation on Particles

When we calculate force on each particle then we need to call function *ForceCal()* for each particle. Calculating force on one particle is totally independent to calculating force on other particles. So we can use this in-dependency for parallelization. While calculating force on particles in CUDA we can calculate force on each particle parallel.

If we have 'n' number of particles then we created $\sqrt{n}$ blocks and $\sqrt{n}$ threads and calculated force on each particle parallel. It is equivalent to calculate force on one particle sequentially and calculate force on every particles parallel. In this method we calculated force parallel by using more blocks and threads and utilized resources of GPU.

3.5 Summary

As mentioned earlier, our goal is to create a compiler which will convert a LINQ code in CUDA code. For this we took an example in this chapter and implemented it in LINQ and then manually wrote code in CUDA for same example. By this we made some observation to convert LINQ operators in CUDA. We will use those observations in next chapter when we will discuss equivalent CUDA code of LINQ operators.

Chapter 4

Technical Details

In this chapter we will show conversion of LINQ code to CUDA code. Previously we took Barnes-Hut algorithm as our example and developed this algorithm first in LINQ and then in CUDA, to show the requirement of a LINQ code in CUDA. Now we will take some small pieces of code for LINQ operators and will show the equivalent code in CUDA.

4.1 CUDA Code for LINQ Operators

Now we will explain in details that how to write a equivalent CUDA code for LINQ operators. It contains basically two parts, first part is same for all operators but second part is dependent on feature of operators.

4.1.1 CUDA Kernel Initialization

In first part we will allocate memory in device by using function *cudaMalloc()* to store array in which we applied operator, and then we will copy data from CPU to GPU by using function *cudaMemcpy()*, and then we will launch kernel. For kernel launch we need to specify the number of threads and blocks that we want to create. As discussed previously in section 2.2.4 CUDA has WARP which contains a group of 32 threads, so better to create 32 threads in each block and if data array size is *n* then we will create *n/32* blocks.

4.1.2 LINQ Operators on CUDA

Till launching the kernel, CUDA code is independent of feature of LINQ operators. After that we will code for second part which is dependent on feature of LINQ operators. Now we will explain this coding in details for each operator.

Projection Operator

Program 4.1 shows an example code for projection operator *select*. In this code a function *func()* is changing each element of array *numbers*. This function take a element from array and return another element of same type. Then We will replace *select* operator by a device function *select_cu()* to convert LINQ operator code in CUDA code.

```
/* some code */
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    var Nums = from n in numbers select func(n);

    foreach (var x in Nums)
            Console.WriteLine(x);
/* some code */
```

Program 4.1: Code for *select* operator.

Now we will see equivalent CUDA code for *select* operator, for this we will create threads equal to size of array. Each thread will call the same function *func()* for one element of array. Here in CUDA each thread will run parallel so run time for function call of all elements will be equal to run time for function call of one element. In program 4.2 we used a variable *tid*, which denotes index of threads. We will map one thread to one element by using this variable.

```
__global__ void select_cu(int *arr, int *size, int *ans)
{
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        while(tid<*size)
        {
                ans[tid] = func(arr[tid]);
                tid += blockDim.x * gridDim.x;
        }
}
```

Program 4.2: Equivalent code for *select* operator in CUDA.

Restriction Operator

Program 4.3 shows a sample code for restriction operator *where*. In this code we are restricting the elements by a function *func()*. This function takes an element of array as input and returns a boolean value. If returned value is *false* then we are restricting that element in result.

```
   /* some code */
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        var Nums = from n in numbers where func(n) select n;

        foreach (var x in Nums)
                Console.WriteLine(x);
   /* some code */
```

Program 4.3: Code for *where* operator.

Equivalent code of *where* operator in CUDA has two parts. In first part we will take another array of same size of input array, let this array is called *restriction* array. This array will be a boolean array, as we have to store a boolean value. In this array the value of i^{th} index will be returned value of function *func()* for i^{th} element of input array. One thread will call function *func()* for one element of array. To map one thread on one element we used a variable *tid*, which represent index of threads as well as index of elements. In second part we will reduce input array by using array *restriction*. We will delete those elements from input array which are having *false* in *restriction* array.

```
__global__ void where_cu(int *arr, int *size, int *restriction)
{
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        while(tid<*size)
        {
                restriction[tid] = func(arr[tid]);
                tid += blockDim.x * gridDim.x;
        }
}
```

Program 4.4: Equivalent code for *where* operator in CUDA.

Partitioning Operator

Program 4.5 shows a sample code for partitioning operator *Take*. We used this operator to take starting k elements from array. Program 4.6 shows equivalent code of program 4.5. When *Take* operator is used in code then we will call device function *Take_cu()* to execute *Take* operator in CUDA.

```
   /* some code */
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        var Nums = numbers.Take(k);

        foreach (var x in Nums)
                Console.WriteLine(x);
   /* some code */
```

Program 4.5: Code for *Take* operator.

To convert LINQ code in CUDA code for a partitioning operator *Take*, we will use another array *take* which will contain the final result. Now we will copy each element from input array to this output array. Copy operation is performed for each element which is having index less than k, where k is input of operator *Take*. Program 4.6 is the final CUDA code for *Take* operator.

```
__global__ void Take_cu(int *arr, int *size, int *take)
{
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        while(tid<*size)
        {
                if(tid<k)
                        take[tid] = arr[tid];
                tid += blockDim.x * gridDim.x;
        }
}
```

Program 4.6: Equivalent code for *Take* operator in CUDA.

Aggregate Operator

Program 4.7 shows a code for aggregate operator *Sum*, which is used to calculate sum of all the elements of input array. Program 4.8 shows equivalent CUDA code of program 4.7, in place of *Sum* operator we will call a device function *sum_cu()*.

```
   /* some code */
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        double numSum = numbers.Sum();

        Console.WriteLine("The sum of the numbers is {0}.", numSum)
            ;
   /* some code */
```

Program 4.7: Code for *Sum* operator.

Now we will explain CUDA code for LINQ operator *Sum*. While we add all the elements of an array we usually take elements from array one by one and add the value of current element in result variable which stores sum of all elements till current element. In this process we are updating value of result variable in each step, so if we run it parallel and one thread will update value for result variable for one element then at same time result variable might be updated for all elements, then we might loose some data. So by this method we can not get correct result.

For aggregate operator *Sum* we used a different technique. In which initially we will divide given array in two equal parts from mid of the array, then we will add $(mid{+}i)^{th}$ element in i^{th} element for each value of i from 0 to *mid*. After this step we need to add only half array. And for remaining array we will apply same technique recursively till remaining array size is 1. This method will take $O(logn)$ time to compute result.

```
const int threadsPerBlock = 32;

__global__ void agg_cu_(int *arr, int *size, int *ans)
{
        __shared__ int cache[threadsPerBlock];
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        int cacheIndex = threadIdx.x;
        int temp = 0;
        while(tid<*size)
        {
                temp += arr[tid];
                tid += blockDim.x * gridDim.x;
        }
        cache[cacheIndex] = temp;
        __syncthreads();
        int i = blockDim.x/2;
        while(i!=0)
        {
                if(cacheIndex<i)
                        cache[cacheIndex]+= cache[cacheIndex+i];
                __syncthreads();
                i/=2;
        }
        if(cacheIndex == 0)
                ans[blockIdx.x] = cache[0];
}
```

Program 4.8: Equivalent code for *Sum* operator in CUDA.

4.2 Issues in Implementation

As we told earlier, we made some observation when coded Barnes-Hut algorithm in CUDA, and then applied those observation to code LINQ operators in CUDA individually. We also faced some problems at that time now we will discuss those problems in details.

4.2.1 NVIDIA Graphic Card Related Issues

In this thesis we implemented Barnes-Hut Algorithm in CUDA. To implement this algorithm we created quadtree and all the functions of this algorithm is executed in this quadtree recursively, so basically most of the function is recursive function. All NVIDIA GPU dose not support recursion. For recursion we need a GPU with compute capability atleast 2.0. At time of writing CUDA code for LINQ operators we want a code which is independent to graphic card, so we did not use recursion at that time.

4.2.2 Memory Allocation in Recursion

As mentioned earlier, functions of Barnes-Hut algorithm are recursive. While creating quadtree by function *CreateTree()* which we discussed in section 3.3.1, we allocated memory to store particles in each recursive call, but GPU has limited memory in it's stack so for large input we did not get result because of lack of stack memory. That is another reason to avoid recursive code in CUDA.

Chapter 5

Result

We saw implementation of Barnes-Hut algorithm in c# using LINQ operators and in GPGPU using CUDA. Here in this section we will show running time of CPU version of code and GPU version of code. Table 5.1 shows the running time difference of CPU code and GPU code for different inputs.

Number of Particles	**CPU**	**GPU**		**Number of Particles**	**CPU**	**GPU**	
	Computation (Sec)	**Computation (Sec)**	**Memory copy (Sec)**		**Computation (Sec)**	**Computation (Sec)**	**Memory copy (Sec)**
100	0.08	0.014	4.037	60000	13.553	0.045	4.25
200	0.084	0.003	4.048	70000	15.965	0.052	4.282
300	0.1	0.009	4.044	80000	18.989	0.063	4.316
400	0.11	0.007	4.045	90000	21.293	0.063	4.353
500	0.12	0.009	4.044	100000	24.546	0.073	4.386
600	0.122	0.01	4.045	200000	55.647	0.14	4.727
700	0.124	0.011	4.045	300000	89.33	0.208	5.056
800	0.136	0.011	4.045	400000	123.236	0.299	5.416
900	0.152	0.011	4.047	500000	159.738	0.354	5.606
1000	0.164	0.011	4.047	600000	193.08	0.435	5.902
2000	0.276	0.01	4.049	700000	232.619	0.432	6.228
3000	0.384	0.0073	4.0537	800000	279.92	0.566	6.475
4000	0.536	0.009	4.056	900000	313.99	0.692	6.802
5000	0.692	0.01	4.058	1000000	347.1	0.574	7.146
6000	0.824	0.008	4.063	2000000	755.46	1.126	10.174
7000	1.008	0.011	4.066	3000000	T.O.	2.119	12.889
8000	1.144	0.01	4.07	4000000	T.O.	2.987	15.95
9000	1.324	0.013	4.072	5000000	T.O.	3.44	18.803
10000	1.488	0.016	4.075	6000000	T.O.	3.947	21.663
20000	3.436	0.021	4.111	7000000	T.O.	4.748	24.704
30000	5.68	0.028	4.148	8000000	T.O.	5.394	27.805
40000	8.137	0.032	4.181	9000000	T.O.	6.386	30.3758
50000	10.581	0.039	4.216	10000000	T.O.	7.018	33.374

Table 5.1: Difference in running time of Barnes-Hut algorithm in CPU and GPU

LINQ code is run on CPU and CUDA code is run on GPU. In table 5.1 there is one column memory copy which shows time taken to copy input data from host to device and to copy back result from device to host. Program 5.1 shows data type Particles. One particle takes 48 bytes of memory. If there are 'n' particles then it will take n*48 bytes of memory and total 2*n*48 byttes of memory will be copied.

```
Location
{
        double x;       // x co-ordinate of Loaction.
        double y;       // y co-ordinate of Loaction.
}

Particles
{
        Location loc;   //Data Type Location:
                            //(x,y) co-ordinates.
        double m;       //mass of Particles.
        double vel;     //velocity of Particles
        double force;   //Forces exerted on Particles
}
```

Program 5.1: Data type Particles.

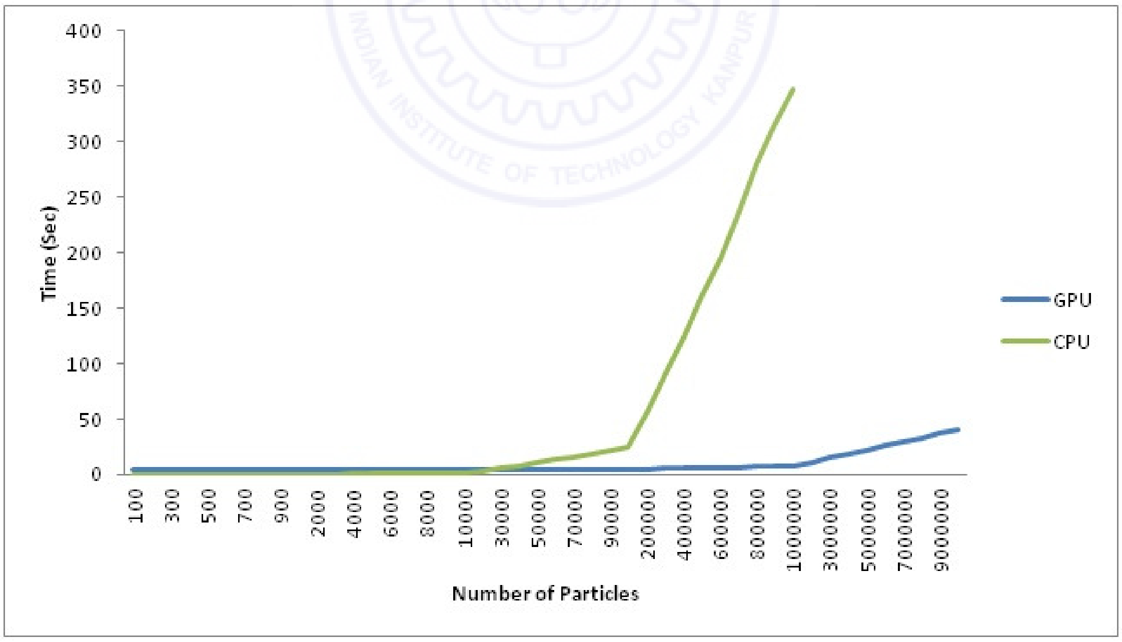


Figure 5.1: Comparison between running time of CPU and GPU including memory copy time.

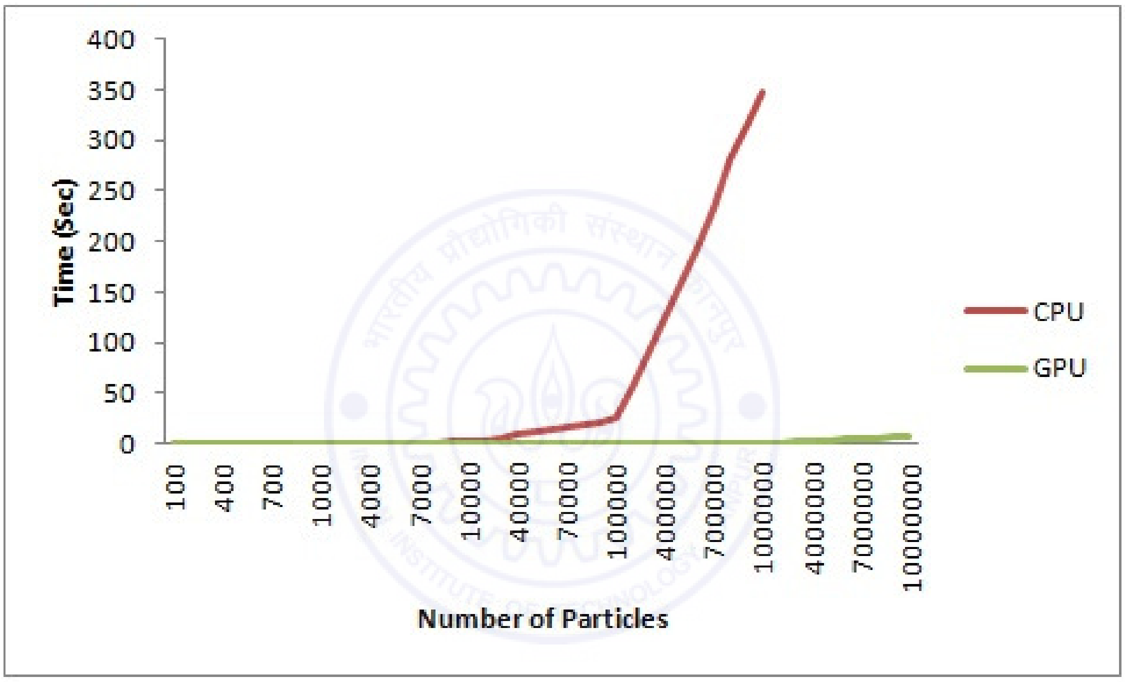


Figure 5.2: Comparison between running time of CPU and GPU excluding memory copy time.

Chapter 6

Related Work

Here in this chapter we will discuss our work with some other techniques. To parallelize a sequential code is a interesting area of research from last many years. Researcher have worked to reduce the human effort required in this process by either completely automatize the process or by encouraging the use of parallel design patterns. Many tools provides ready made templates to write a parallel code. Many researches have been done to parallelize some loops of code or some portion of code. Some tools generate finale executable file which contains parallel constructs. These tools work well for parallelization and understood semantic of popular benchmarked code. some of those tools and techniques are discussed below.

6.1 DryadLINQ

DryadLINQ [Dry12] is a system and a set of language extensions that enable a new programming model for large scale distributed computing. This is a reference of LINQ model to write massive data parallel programs. DryadLINQ uses LINQ programming and converts it into data parallel programming using standard .NET development tools. DryadLINQ runs on Dryad execution engine. A DryadLINQ system automatically translate a data parallel code into a distributed execution environment which runs on dryad execution platform.

6.2 Parallel Database

Parallel Database [PDS92] is a research towards parallel execution of queries. In this research database is distributed in different disks and query is executed parallel on those different disks by many processors. This distribution of data is called *Data Partitioning*. Data partitioning can be done by many ways like range partition, round-robin partition, hashing partition. Many projects are implemented on this research, such as Gamma [Gam90], Bubba [Bub90] and Volcano [Val90]. Some of commercial products is also developed for data warehousing like Teradata, IBM DB2 Parallel Edition [DPE95] and Tandem SQL/MP [Tan95].

In our work we also used data partition mechanism to parallelize query execution, but we used CUDA for our work and divided data in different blocks and threads of CUDA. Our implementation is also similar to hashing data partition because we mapped each data element to a thread by their index of array.

6.3 Large Scale Data-Parallel Computation Infrastructure

In last decade the architecture of database is changed to very large datasets. for fast execution of this type of datasets we need high performance computing. One of the earliest commercial generic platforms for distributed computation was the Teoma Neptune platform [TeNe03], which introduced a MapReduce computation paradigm inspired by MPIs Reduce operator. In Google MapReduce framework [MR04] computational model is extended to separate the execution layer from storage, and virtualized the execution. The Hadoop open-source port of MapReduce uses this architecture. A grid-based architecture for execution layer is proposed by NetSolve [Net05]. At storage level many very large scale simple database developed such as Google's BigTable [BigT06], Amazon's Simple DB and Microsoft SQL Server Data Services.

Here in our implementation we are copying data from main memory to CUDA device memory. So we can use any of these storage for our work which can give data in simple array form in main memory. After that we can copy data to device memory and run device code parallel.

6.4 Databases on GPU

There has been lots of research to speed up query languages on GPU. In one research SQL operations are mapped into GPU [SQL10]. In this research, they implemented some of the function of SELECT query in GPU. For which they used SQLite database to execute queries and to switch data between CPU and GPU. SQLite virtual machine is reimplemented as CUDA kernel to execute queries on GPU.

Many other researches are also there in which GPU is used to execute queries. These researches use preventives such as Sort and Scatter, that can be combined and run in succession on the same data to produce the results of common database queries. In one research database queries is divided into predicate evaluation, boolean combination, and aggregation functions [DBq05]. In another work, binary searches, p-ary searches [PSea08], tree operations, relational join operations [RJop07] is included.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Now a days database is becoming very large. Queries run on terabytes of data. In this kind of database to run query faster multi-cores are not enough. For better speed-up researches are moving from multi-cores to GPU. So our idea is also in this direction which is to present the use of GPU in execution of LINQ and to give some techniques to implement a compiler which will automatically generate a CUDA code for given LINQ code.

We took a well know problem Barnes-Hut algorithm for our implementation and implement it in LINQ and then CUDA. Then by use of this implementation in two different environment we wrote LINQ operators in CUDA. These CUDA code of LINQ operators are just to show that LINQ operators can be converted in CUDA. By use of these CUDA code we told how can we write a compiler to convert a LINQ Code in CUDA code. So the main aim behind this work is to help to write compiler.

7.2 Future Work

7.2.1 Development of Tool

As we told earlier our work is in a direction to make a tool which will convert LINQ code to CUDA code, but tool is not yet built. Here we are giving a knowledge to build that tool. Whatever we discussed in chapter 4 is basically a starting point to build this tool. The tool could be semi-automatic or full automatic. In semi-automatic we can provide a framework which will take LINQ operators as input and will return CUDA code. We can design this framework for CUDA which will write most of the code in CUDA but for LINQ operators it will take it direct and give equivalent CUDA code.

In other way we can directly make full automatic compiler which will take LINQ code as input and return CUDA code.

7.2.2 Optimization of Tool

The other kind of work which we can do in this tool is to optimize it so that it will generate efficient number of threads and blocks so that running time will be efficient. We can also add some features in this tool for multiple GPU cards.

Bibliography

[BH86] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, December 1986.

[SQL10] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. *GPGPU-3, Pittsburg, PA, USA*, 2010.

[DPE95] A. Goyal H. Hsiao A. Jhingran S. Padmanabhan G.P. Copeland C.K. Baru, G. Fecteau and W.G. Wilson. Db2 parallel edition. *IBM Systems Journal 34*, February 1995.

[PDS92] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database processing. *Communicationsof the ACM 36*, January 1992.

[Dry12] DryadLINQ, March 2012. `http://research.microsoft.com/en-us/projects/dryadlinq/`.

[BigT06] S. Ghemawat W.C. Hsieh D.A. Wallach M. Burrows T. Chandra A. Fikes F. Chang, J. Dean and R.E. Gruber. Bigtable: A distributed storage system for structured data. *In Symposium on Operating System Design and Implementation (OSDI), 2006.*, 2006.

[HPF97] High Performance Fortran Forum. High performance fortran language specification version 2.0. Technical report, Rice University, 1997.

[GPG12] GPGPU: `http://gpgpu.org/`, March 2012.

[Val90] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD International Conference on Management of data.*, 1990.

[Has12] Haskell, March 2012. `http://www.haskell.org/haskellwiki/Haskell`.

[Bub90] L. Clay G. Copeland S. Danforth M. Franklin B. Hart M. Smith H. Boral, W. Alexander and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering.*, January 1990.

[Gam90] Donovan Schneider Hui-I Hsiao Allan Bricker J. DeWitt, Shahram Ghandeharizadeh and Rick Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering.*, January 1990.

[MR04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Sixth Synmposium on Operatiing System Design and Implementaion (OSDI'04), San Francisco*, December 2004.

[TeNe03] T. Yang L. Chu, H. Tang and K. Shen. Optimizing data aggregation for cluster-based internet services. *Symposium on Principles and practice of parallel programming (PPoPP)*, 2003.

[LIN12a] 101 LINQ samples, March 2012. `http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b`.

[LIN12b] LINQ, March 2012. `http://msdn.microsoft.com/en-us/library/bb397926.aspx`.

[Net05] J. Dongarra M. Beck and J.S. Plank. Netsolve: A massively parallel grid execution system for scalable data intensive collaboration. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[DBq05] W. Wang M. Lin N.K. Govindaraju, B. Lloyd and D. Manocha. Fast computation of database operations using graphics procesoors. *SIGGRAPH05: ACM SIGGRAPH 2005 Courses, page 206, New York, NY, USA*, 2005.

[NVI11] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture, C Programming Guide, Version 4.1.* 2011.

[Ope12] OpenCL: `http://www.khronos.org/opencl/`, March 2012.

[RJop07] M. Lu K. Yang N.K. Govindaraju Q. Luo R. Fang, B. He and P.V. Sander. GPUQP: Query co-processing using graphics processors. *ACM SIGMOD International Conference on Management of Data, pages 10611063, New York, NY, USA,*, 2007.

[Tan95] R. Glasstone S. Englert and W. Hasan. Parallelism and its price : A case study of nonstop SQL/MP. *Sigmod Record*, 1995.

[PSea08] A. Di Blas T. Kaldeway, J. Hagen and E. Sedlar. Parallel search on video cards. 2008. Technical report, Oracle.