**King Abdulaziz University**

**Faculty of** **Computing and**

**Information Technology**

**Department of Computer Science**

**CPCS-499 Project Progress Report Part 1**

**Approximate Query Location in Mobile Computing**

**(Using GPU)**

**Supervisor: Dr.Jonathan (Yahya) Cazalas**

**Signature:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Member** | **Name** | **ID** | **E-Mail** |
| **1** | **Ahmad Ghassan Hariri** | **1008381** | **ahmadgjh@gmail.com** |
| **2** | **Abdullah Salem Khered** | **1008623** | **Khered20@gmail.com** |

**Table of Contents**

Contents

[Summary of previous work 3](#_Toc384227313)

[Aim of the project 3](#_Toc384227314)

[Problem definition 3](#_Toc384227315)

[Project scope 3](#_Toc384227316)

[Expected outcome 3](#_Toc384227317)

[Target Users 3](#_Toc384227318)

[Suggested solution 4](#_Toc384227319)

[Functional Requirements :- 4](#_Toc384227320)

[Nonfunctional Requirements :- 4](#_Toc384227321)

[Data Requirements :- 4](#_Toc384227322)

[Hardware :- 4](#_Toc384227323)

[Software :- 4](#_Toc384227324)

[Methods and Approaches 5](#_Toc384227325)

[Planning 5](#_Toc384227326)

[Analysis 5](#_Toc384227327)

[Design 5](#_Toc384227328)

[Implementation 5](#_Toc384227329)

[Maintenance 5](#_Toc384227330)

[Activities and Actions 6](#_Toc384227331)

[Completed : 6](#_Toc384227332)

[Not Completed : 6](#_Toc384227333)

[Implementation 7](#_Toc384227334)

[Tools & technologies 7](#_Toc384227335)

[Interface 8](#_Toc384227336)

[Code 11](#_Toc384227337)

[References 18](#_Toc384227338)

[Research papers : 18](#_Toc384227339)

[Book : 18](#_Toc384227340)

# Summary of previous work

## Aim of the project

The aim is to learn parallel programming on GPUs using NVidia’s CUDA and use this knowledge to solve problems & improve the evaluations in the domain of real-time approximate query location in mobile computing.

## Problem definition

The problem is about a big number of objects and those objects are moving randomly. All those objects send their locations in real-time to the server. Also, those objects want to know where their neighbors.

We tackle a challenge in processing continuous range queries: coping with data uncertainty inherently associated with location data (stale, inaccurate position).

These queries must be evaluated in REAL-TIME. Parallel programming on the GPU provide a great solution to evaluate these queries.

## Project scope

The scope of this project is:

* Learn CUDA C.
* Make simulations in the GPU & CPU to solve approximate queries.
* Improve the evaluation for the approximate query locations.
* Compare the time results between the CPU & the GPU.

## Expected outcome

CUDA program that parallelize the execution of large number of queries for finding ( the neighbors ) with approximate locations.

## Target Users

The target users are large environments in which needs to be improved by querying to find approximate location to theirs neighbors in order to achieve their goals (avoid causing accidents, find, manage, manipulate,... etc.).

## Suggested solution

First, we will create four simulations in CPU & GPU to model the environment – movement- by giving a coordinate to every object and letting it move randomly.

In the CPU we will make two simulations:

* Simulation to solve range queries using exact locations.
* Simulation to solve range queries using approximate locations.

Also, In the GPU we will make the same simulations to improve the evaluation.

## Functional Requirements :-

* Simulate an environment with continuous moving objects.
* Find exact & approximate locations by an algorithm.
* Compare & evaluate the performance between the CPU & the GPU.

## Nonfunctional Requirements:-

* Response time.
* Throughput.
* Reliability.
* Maintainability.

## Data Requirements:-

The xCoordinate & yCoordinate for every object and the query range.

## Hardware:-

GPU Nvidia card supported CUDA.

## Software:-

* Microsoft visual studio ( C Language ) .
* Windows 8 .
* CUDA SDK 5.5 .
* CUDA Device Driver.
* NVIDIA NSight VISUAL STUDIO Edition Tool .

# Methods and Approaches

## Planning

We planned for the project by work break down structure (WBS) then scheduled the project by Gantt Chart & time estimates to manage & organize our work in order to complete the project.

## Analysis

We started by reading about the problem in research papers and read a book teach us the CUDA language.

Then, We defined the project scope , functional requirements , non-functional requirements and the expected outcome.

## Design

We designed an initial design for the project , flow chart , use case diagram and class diagram. Then ,

## Implementation

we started coding the project . and the coding is divided into 6 parts :

1. Simulate an environment with continuously moving objects.
2. Find exact location by an algorithm in CPU.
3. Find exact location by an algorithm in GPU.
4. Find approximate location by an algorithm in CPU.
5. Find approximate location by an algorithm in CPU.
6. Compare & asses the results between the CPU & the GPU.

## Maintenance

After completing the code, we will test the program in order to remove the errors.

# Activities and Actions

## Completed:

* Simulate an environment with continuously moving objects.
* Find exact locations by an algorithm in CPU.

## Not Completed:

* Find exact locations by an algorithm in GPU.
* Find approximate locations by an algorithm in CPU.
* Find approximate locations by an algorithm in GPU.
* Compare & evaluate the performance between the CPU & the GPU.

The completed part is the major part and the remaining part is small comparing to it (we will use the CPU code then modify the query function in order to use GPU ).

# Implementation

## Tools & technologies

**Hardware:-**

We need GPU NVidia card supports CUDA in order to do parallel computations.

**Software:-**

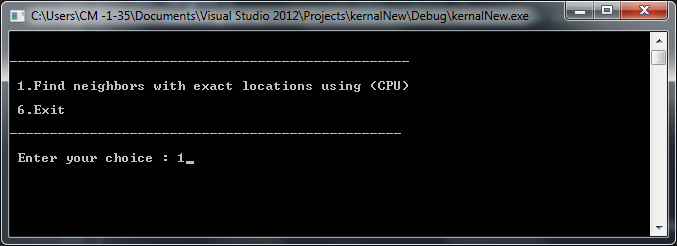
* Microsoft visual studio ( C Language ) , C is an interface to the CUDA platform.
* Windows 8 .
* CUDA SDK 5.5 .
* CUDA Device Driver.
* NVIDIA NSight Visual Studio Edition Tool , it brings parallel programming on the (GPU) into visual studio.
* Microsoft Visio ( To design the project ).

## Interface

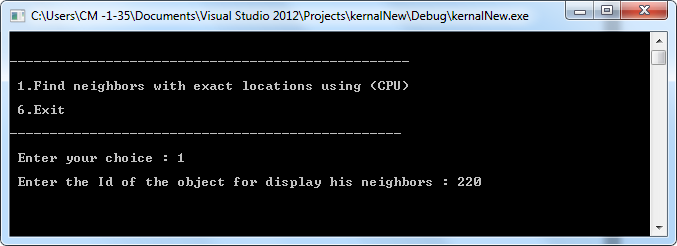
The interface is a command line interface ( because the project is simulation-based type ).

The interface:

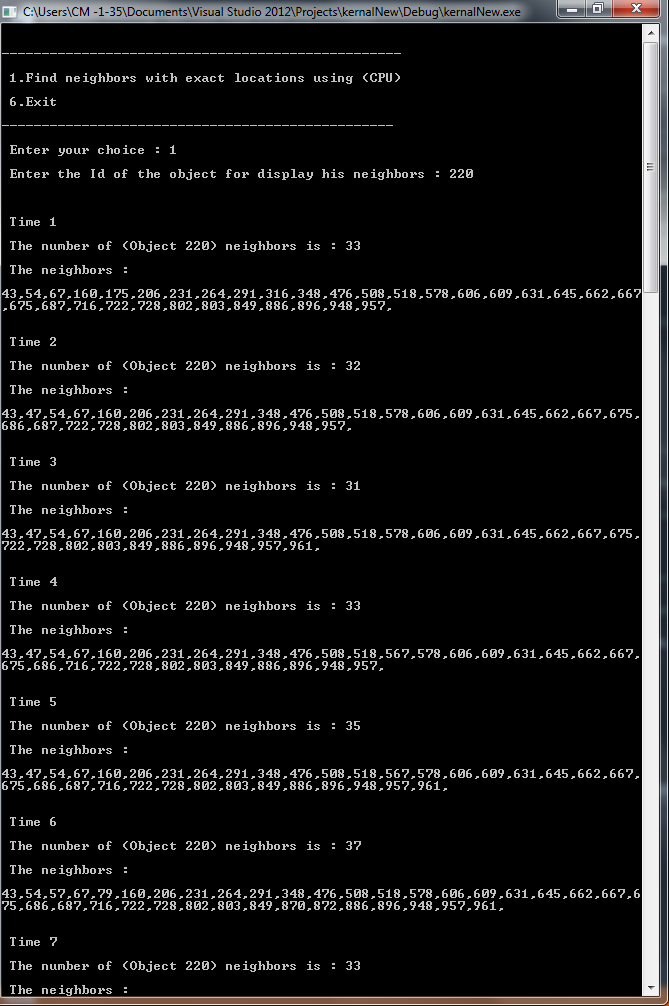
* The user need to enter a choice from the list.

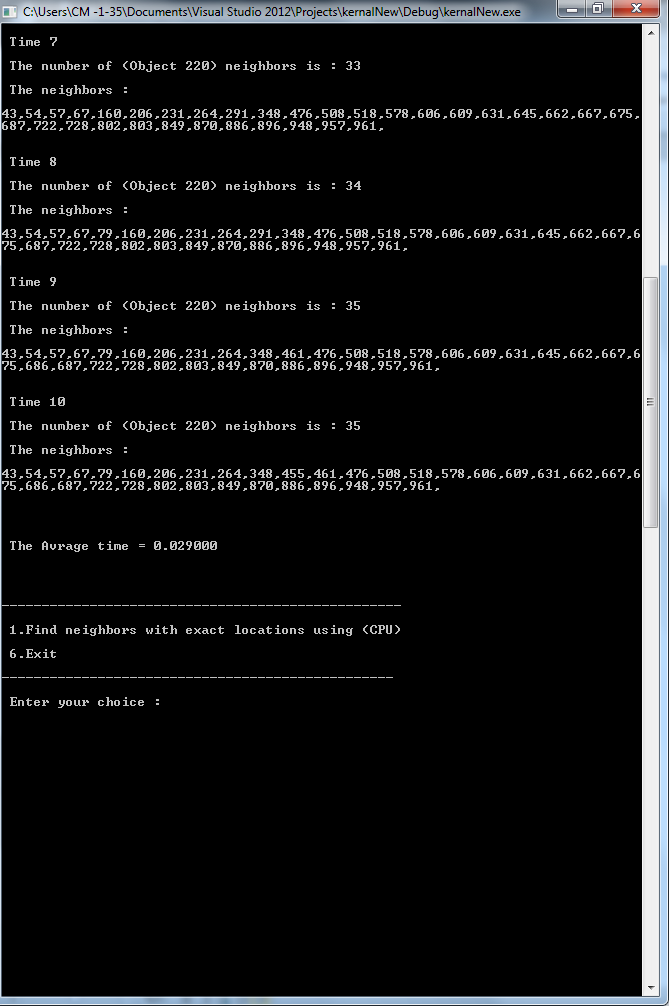


* The user needs to type the ID of the object in order to display the neighbors.



* The results (neighbors) & the average time for the execution of the query.





## Code

The code:

//File : Exact location in CPU & GPU

//Develpoers : Abdullah Salem Khered - Ahmad Jamal Hariri 1008381

// Libraries used

#include<stdlib.h>

#include <stdio.h>

#include <time.h>

#include <math.h>

#include <cuda.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

// Definning identifires

#define gridSizeX 1000 // Environment x-axis size

#define gridSizeY 1000 // Environment y-axis size

#define numOfObjects 1000 // The number of objects at the environmet

#define queryDistRange 100 // Query range in x-axis & y-axis

#define simuIteration 10 // Simulation iterations (number of simulation iterations)

#define objMovmentRange 10 // Maximum movment range in the x-axis & y-axis

#define imin(a,b) (a<b?a:b)

const int threadsPerBlock = 512;

const int blocksPerGrid =imin( 100, (numOfObjects+threadsPerBlock-1) / threadsPerBlock );

// User defined Type

typedef struct object // Define a type called object

{

int \*neighbors; // Each neighbor ID

int numOfNeighbors; // The number of neighbors for each object

} ;

//=========================================Functions===============================================

// Display the program menu

int showMenu()

{

int choice = 0;

printf("\n\n--------------------------------------------------\n\n 1.Find neighbors with exact locations using (CPU) \n\n 6.Exit \n\n-------------------------------------------------\n\n Enter your choice : ");

scanf("%d",&choice); // Store the input in the variable

return (choice); // Return the value typed by the user

}

//==================================================================================================

// Initialize objects positions (random coordinates)

void setObjectsPositions (struct object\* objects,int\*x,int\*y)

{

int i;

srand((unsigned)time(0));

for (i = 0; i < numOfObjects; i++)

{

x[i] = rand() % gridSizeX + 1;

y[i] = rand() % gridSizeY + 1;

objects[i].neighbors = (int\*) malloc( 4 \* sizeof(int));

objects[i].numOfNeighbors = 0;

}

}

//==================================================================================================

// Remove (stored) nighbors for each object

void clearNeighbors (struct object\* objects)

{

int i;

for (i = 0; i < numOfObjects; i++)

{

free(objects[i].neighbors);

objects[i].neighbors = (int\*) malloc( 4 \* sizeof(int));

objects[i].numOfNeighbors = 0;

}

}

//==================================================================================================

// Move the objects at random x & y coordinates

void moveRandomly(struct object\* objects,int\*x,int\*y){

int i, moveX, moveY;

for (i = 0; i < numOfObjects; i++)

{

moveX = (rand() % (objMovmentRange \* 2 + 1)) - 10;

moveY = (rand() % (objMovmentRange \* 2 + 1)) - 10;

if (x[i] + moveX > gridSizeX || x[i] + moveX < 0 || y[i] + moveY > gridSizeY || y[i]+moveY < 0)

{

continue;

}

x[i] += moveX;

y[i] += moveY;

}

}

//==================================================================================================

// Adding neighbors for each object (if the neighbor(s) exists within the specified range)

void addNeighbors (struct object\* objects,int i,int j)

{

if((objects[i].numOfNeighbors)>0 && objects[i].numOfNeighbors%4==0)

{

objects[i].neighbors=(int\*)realloc(objects[i].neighbors,(objects[i].numOfNeighbors+4)\*sizeof(int));

}

objects[i].neighbors[objects[i].numOfNeighbors]=j;

objects[i].numOfNeighbors++;

if((objects[j].numOfNeighbors)>0 && objects[j].numOfNeighbors%4==0)

{

objects[j].neighbors=(int\*)realloc(objects[j].neighbors,(objects[j].numOfNeighbors+4)\*sizeof(int));

}

objects[j].neighbors[objects[j].numOfNeighbors]=i;

objects[j].numOfNeighbors++;

}

//==================================================================================================

// Query to find for each object all the neighbors that falls within the range ( CPU )

void queryExact(struct object\* objects,int \*x,int\*y) {

int i, j;

double euclideanDistance, dist;

for (i = 0; i < numOfObjects; i++)

{

for (j = i+1; j < numOfObjects; j++)

{

dist = (x[i] - x[j]) \* (x[i] - x[j]) + (y[i] - y[j]) \* (y[i] - y[j]) ;

euclideanDistance = sqrt(dist) ;

if (euclideanDistance <= queryDistRange)

{

addNeighbors(objects,i,j);

}

}

}

}

//==================================================================================================

// Query to find for each object all the neighbors that falls within the range ( GPU )

\_\_global\_\_ void queryExactCUDA(int id,int \*x,int \*y,

int\* destinationArray,int\*d\_nipNumber)

{

int j = threadIdx.x + blockIdx.x \* blockDim.x;

while (j < numOfObjects) {

if (sqrt((double)((x[id] - x[j]) \* (x[id] - x[j]) + (y[id] - y[j]) \* (y[id] - y[j]))) <= queryDistRange)

{

destinationArray[d\_nipNumber[0]++]=j;

}

j += blockDim.x \* gridDim.x;

}

}

//==================================================================================================

// Query to find for each object all the neighbors that falls within the range (GPU )

void queryExactGPU(struct object\* objects,int \*x,int \*y) {

int \*d\_x ; // Array for x coordinates for the objects

int \*d\_y ; // Array for y coordinates for the objects

int\* d\_destinationArray,\*d\_nipNumber; // the device array

int h\_nipNumber[1];

cudaMalloc((void\*\*)&d\_destinationArray,numOfObjects\*sizeof(int)); //allocate the device memory for destination array

cudaMalloc((void\*\*)&d\_nipNumber,sizeof(int));

cudaMalloc((void\*\*)&d\_x, numOfObjects \* sizeof(int));

cudaMalloc((void\*\*)&d\_y, numOfObjects \* sizeof(int));

cudaMemcpy( d\_x,x,numOfObjects\* sizeof(int),cudaMemcpyHostToDevice );

cudaMemcpy( d\_y,y,numOfObjects\* sizeof(int),cudaMemcpyHostToDevice );

for (int i = 0; i < numOfObjects; i++)

{

size\_t id=i;

h\_nipNumber[0]=0;

cudaMemcpy( d\_nipNumber,h\_nipNumber,sizeof(int),cudaMemcpyHostToDevice );

queryExactCUDA<<<blocksPerGrid, threadsPerBlock>>>(id,d\_x,d\_y, d\_destinationArray,d\_nipNumber);

cudaMemcpy(h\_nipNumber,

d\_nipNumber,

sizeof(int),

cudaMemcpyDeviceToHost);

objects[i].numOfNeighbors=h\_nipNumber[0];

objects[i].neighbors=(int\*)realloc(objects[i].neighbors,(objects[i].numOfNeighbors)\*sizeof(int));

cudaMemcpy(objects[i].neighbors,

d\_destinationArray,

objects[i].numOfNeighbors\*sizeof(int),

cudaMemcpyDeviceToHost);

}

cudaFree(d\_x);

cudaFree(d\_y);

cudaFree(d\_destinationArray);

}

//======================================================

void print(struct object \*objects,int objectChoice) {

int i;

printf( " The number of (Object %d) neighbors is : %d\n\n The neighbors :\n\n",objectChoice,objects[objectChoice].numOfNeighbors);

for (i = 0; i < objects[objectChoice].numOfNeighbors; i++)

{

printf("%d,",objects[objectChoice].neighbors[i]);

}

}

//==================================================================================================

// Main function

int main(){

float avgTime;

int i, j, iteration, choice,objectChoice;

double start, end;

int x[numOfObjects],y[numOfObjects];

struct object objects[numOfObjects]; // Variable from the declared type called objects

srand((unsigned)time(0));

setObjectsPositions(objects,x,y); // Initialization of objects positions

do{

choice = showMenu(); // Show the menu to the user

if (choice == 1) // Find neighbors with exact locations

{

avgTime = 0; // initialize average time to 0

printf("\n Enter the Id of the object for display his neighbors : "); // Display detailed information about the results

scanf("%d",&objectChoice);

for (iteration = 1; iteration <= simuIteration; iteration++) // Start the simulation

{

clearNeighbors(objects); // To clear the objects neighbors when the program restart

moveRandomly(objects,x,y); // Move the objects at random x & y coordinates

printf("\n\n\n Time %d\n\n",iteration);

start=clock(); // Start recoding query time

queryExact(objects,x,y); // Find neighbors with exact locations

end=clock(); // Stop recoding query time

print(objects,objectChoice);

avgTime += (end-start) / CLK\_TCK; // Calculate the time in seconds

}

printf("\n\n\n\n\n The Avrage time = %f\n\n\n",(avgTime / simuIteration)); // Print the average time

}

else if (choice == 2) // Find neighbors with exact locations

{

avgTime = 0; // initialize average time to 0

// Display detailed information about the results

printf("\n Enter the Id of the object to display his neighbors : ");

scanf("%d",&objectChoice);

for (iteration = 0; iteration < simuIteration; iteration++) // Start the simulation

{

clearNeighbors(objects); // To clear the objects neighbors when the program restart

moveRandomly(objects,x,y); // Move the objects at random x & y coordinates

printf("\n\nTime %d\n\n",iteration);

start=clock(); // Start recoding query time

queryExactGPU(objects,x,y); // Find neighbors with exact locations

end=clock(); // Stop recoding query time

print(objects,objectChoice);

avgTime += (end-start) / CLK\_TCK; // Calculate the time in seconds

}

printf("\n\n The Avrage time = %f\n\n",(avgTime / simuIteration)); // Print the average time

}

getchar(); // To prevent the program from closing

} while (choice != 5); // If 5 then exit program , else execute the program again

} // End of main function

//=======================================END OF CODE================================================

# References

## Research papers:

1. A. Aved, K. A. Hua, and A. Petkova, “Processing Approximate Moving Range Queries in Mobile Sensor Environments”, CSE-09 international conference, 2009 – IEEE .
2. A. Aved, K. A. Hua, and A. Petkova, “CAWA: Continuous Approximate Where-About Queries” - Scalable Information Systems, 2009 – Springer.

## Book:

1. “CUDA by Example: An Introduction to General-Purpose GPU Programming”, J. Sanders, E. Kandrot, NVIDIA Corp, 2010.