|  |
| --- |
| Parallel Implementation of Connected Component Labelling in NVIDIA® CUDA (Compute Unified Device Architecture) |
| CSE Main-Project 8th Semester |
| Souham Biswas, B.Tech. Computer Science & Engineering Roll No. – 25 |

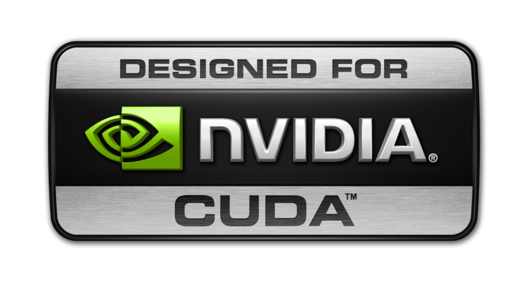


Table of Contents

[INTRODUCTION 2](#_Toc420393796)

[ACKNOWLEDGEMENT 3](#_Toc420393797)

[CERTIFICATE 4](#_Toc420393798)

[SOFTWARE PRE-REQUISITES 5](#_Toc420393799)

[SOFTWARE OVERVIEW 6](#_Toc420393800)

[MAIN FUNCTIONAL OUTLINE 8](#_Toc420393801)

[FEATURES 10](#_Toc420393802)

[TECHNOLOGIES AND SOFTWARES USED 10](#_Toc420393803)

[SOFTWARE WORKINGS 13](#_Toc420393804)

[Image Smoothening- 13](#_Toc420393805)

[Thresholding- 14](#_Toc420393806)

[Connected Component Counting- 14](#_Toc420393807)

[CODE 15](#_Toc420393808)

[Development Softwares Required 34](#_Toc420393809)

[CONCLUSION 35](#_Toc420393810)

[BIBLIOGRAPHY 36](#_Toc420393811)

[EXAMINER’S REMARKS 37](#_Toc420393812)

# INTRODUCTION

**Overview –**

This report is a documentation of the workings of an NVIDIA® CUDA (Compute Unified Device Architecture) based application which implements a parallelized version of the Connected Component Labelling algorithm on an NVIDIA® Graphics Processing Unit (GPU) alongwith various other image smoothening algorithms to achieve significant speedups. This includes details about running times and speedup gains achieved by parallelizing traditional serial CCL (Connected Component Labelling) algorithms.

**Motivation and Background –**

Connected-component labeling (alternatively connected-component analysis, blob extraction, region labeling, blob discovery, or region extraction) is an algorithmic application of graph theory, where subsets of connected components are uniquely labeled based on a given heuristic.

Connected-component labeling is used in computer vision to detect connected regions in binary digital images, although color images and data with higher dimensionality can also be processed. When integrated into an image recognition system or human-computer interaction interface, connected component labeling can operate on a variety of information. Blob extraction is generally performed on the resulting binary image from a thresholding step. Blobs may be counted, filtered, and tracked.

Considering the prominence of Connected Component Labelling and its applications, an approach towards speeding up its execution has been made by a parallel GPU based implementation.

# ACKNOWLEDGEMENT

I would like to thank Prof. R.R. Tewari for his invaluable guidance provided throughout the project development.

Also, this project would have been impossible without the NVIDIA CUDA programming tutorials provided by Udacity.com and taught by Prof. John Owens, University of California, Davis and David Luebke, Senior Director of Research, NVIDIA®.

Lastly, but not the least, I thank everyone else involved directly or indirectly with the development of this software, as this page is too short to list everyone.

# CERTIFICATE



This is to certify that **Mr. Souham Biswas** has successfully prepared and completed the project under my direct and close supervision and that this is a bonafide piece of work done by him.

**Class:** Bachelor of Technology , 8th semester

**Branch:** Computer Science Engineering

**Academic Year:** 2014-15

**Institution Name:** J.K. Institute of Applied Physics & Technology

Signature of Examiner: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# SOFTWARE PRE-REQUISITES

* **NVIDIA® CUDA-Enabled Graphics Processing Unit (GPU)-**

A CUDA enabled GPU is pre-requisite to run CUDA based applications which utilize the GPU as a GPGPU (General Purpose Graphics Processing Unit). A GPGPU is able to perform general computational tasks on the graphics hardware apart from graphics-related ones.

* **NVIDIA® CUDA Runtime**

A CUDA runtime is required for the CUDA APIs to interface with the GPU. It is also required by the CUDA-C compiler to map the C code to the instruction set of the GPU residing on the system.

# SOFTWARE OVERVIEW

This software is basically an example to demonstrate the reduction in execution time which can be achieved by parallelizing traditional serial Connected Component Labelling Algorithms.

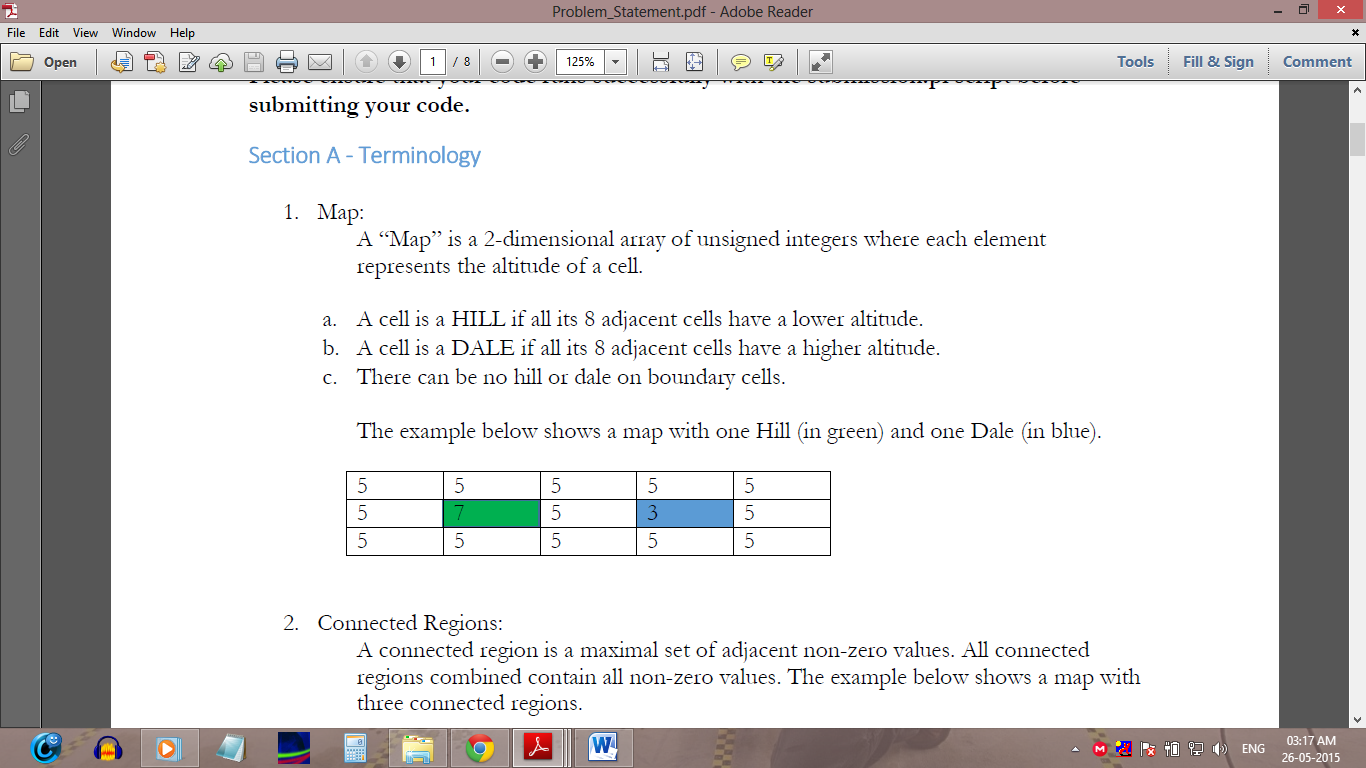
Defined below are the terminologies specific to this application –

1. **Map-**

A “Map” is a 2-dimensional array of unsigned integers where each element represents the altitude of a cell.

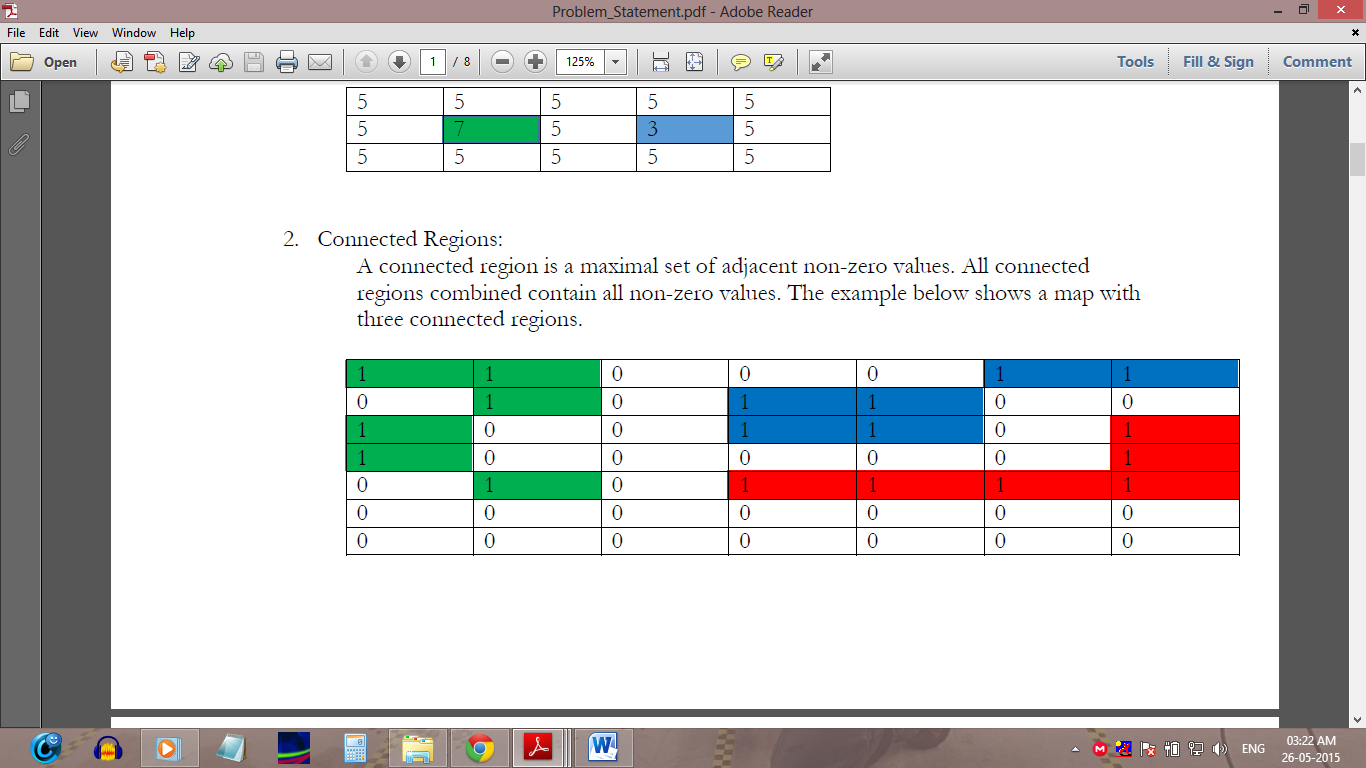
* + 1. A cell is a HILL if all its 8 adjacent cells have a lower altitude.
    2. A cell is a DALE if all its 8 adjacent cells have a higher altitude.
    3. There can be no hill or dale on boundary cells.

The example below shows a map with one Hill (in green) and one Dale (in blue).



1. **Connected Regions-**

A connected region is a maximal set of adjacent non-zero values. All connected regions combined contain all non-zero values. The example below shows a map with three connected regions.



1. **Mean of Surrounding Cells-**

The mean of the surrounding cells is the arithmetic average of the surrounding cells, computed by summing all the values and then dividing by the count of cells. For the map below, we can compute the mean of the cells surrounding X as follows:

|  |  |  |
| --- | --- | --- |
| A | B | C |
| D | X | E |
| F | G | H |

1. **Median of Surrounding Cells-**

The median of the surrounding cells is defined as the arithmetic average of the middle values in the sorted list of values. For the map below, we can compute the median of the cells surrounding X as follows:

|  |  |  |
| --- | --- | --- |
| A | B | C |
| D | X | E |
| F | G | H |

Compute median as-

Sort the surrounding cells and,

## MAIN FUNCTIONAL OUTLINE

The main working modules of the application are enumerated and described below-

Thresholding

Bump Removal

Smoothened Map

Binary Map

Number of Connected Components

MAP

Connected Component Counting

* **Map Processing**

This module deals with processing of the map to a stage where it can be readily passed on for connected component counting–

* **Bump Removal**

This is concerned with replacing every “hill” with the surrounding cells’ average and every “dale” with the surrounding cells’ median as described previously.

* **Thresholding**

Thresholding is done on the smoothened image to generate a binary image containing the “blobs” or regions of pixels of same values whose count is to be found out. It is done by calculating the average of all the altitudes from the smoothened map and subsequently replacing each cell with a ‘0’ or a ‘1’ based upon whether the cell value is lesser or greater than the average value respectively.

* **Connected Component Counting**

This module takes a binary image as input and outputs the number of “blobs” or cone ted component regions in it. It consists of the following parts-

* **Stamping**

This routine basically stamps collections of 4 pixels related by 4-way connectivity with the index number which sequentially identify each such block.

* **Stitching**

This routine “stitches” together different regions having different stamps but are adjacent to each other. The higher stitch number is replaced by the lower onw.

* **Count Determination**

The final count of the connected component regions is determined from the maximum number remaining on the map after the stamping and stitching processes are complete as explaine previously.

## FEATURES

**I. Speed-** The software since it utilizes CUDA, will have a typical speedup of upto 3.2 when compared to a normal uniprocessor (SISD) system.

**II. Scalable-** The application is scalable across all Windows® platforms which also have a CUDA enabled NVIDIA® GPU.

**III. Wide Prominence-** Connected Component Labelling finds a lot of application in popular areas like medical image analysis, space image analysis etc. The modules developed as a part of this program may be directly deployed as a part of any application.

## TECHNOLOGIES AND SOFTWARES USED

**CUDA-C Programming Language –**

CUDA stands for Compute Unified Device Architecture. It is a programming paradigm developed by NVIDIA® which includes libraries and a compiler (nvcc compiler). This technology allows developers to harness the massively parallel processing capabilities of the GPU.

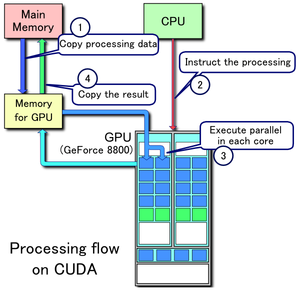
The language “CUDA-C” is a minor modification of the much popular C-language. Program constructs known as ‘kernels’ have been introduced which form the building blocks of any CUDA code. The kernels are basically threads which are to be executed in parallel on the different execution units (known as Streaming Multiprocessors) present on an NVIDIA® GPU.

**CUDA Enabled Graphics Processing Unit (GPU) –**

The GPU, as a specialized processor, addresses the demands of real-time high-resolution 3D graphics compute-intensive tasks. As of 2012, GPUs have evolved into highly parallel multi-core systems allowing very efficient manipulation of large blocks of data. A CUDA Enabled GPU is a GPU starting from the range of NVIDIA® GeForce 8X series. CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

* Scattered reads – code can read from arbitrary addresses in memory
* Unified virtual memory (CUDA 4.0 and above)
* Unified memory (CUDA 6.0 and above)
* Shared memory – CUDA exposes a fast shared memory region that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
* Faster downloads and readbacks to and from the GPU.
* Full support for integer and bitwise operations, including integer texture lookups.

Illustrated below, is an example of a CUDA processing flow -



1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem

**CUDA Runtime –**

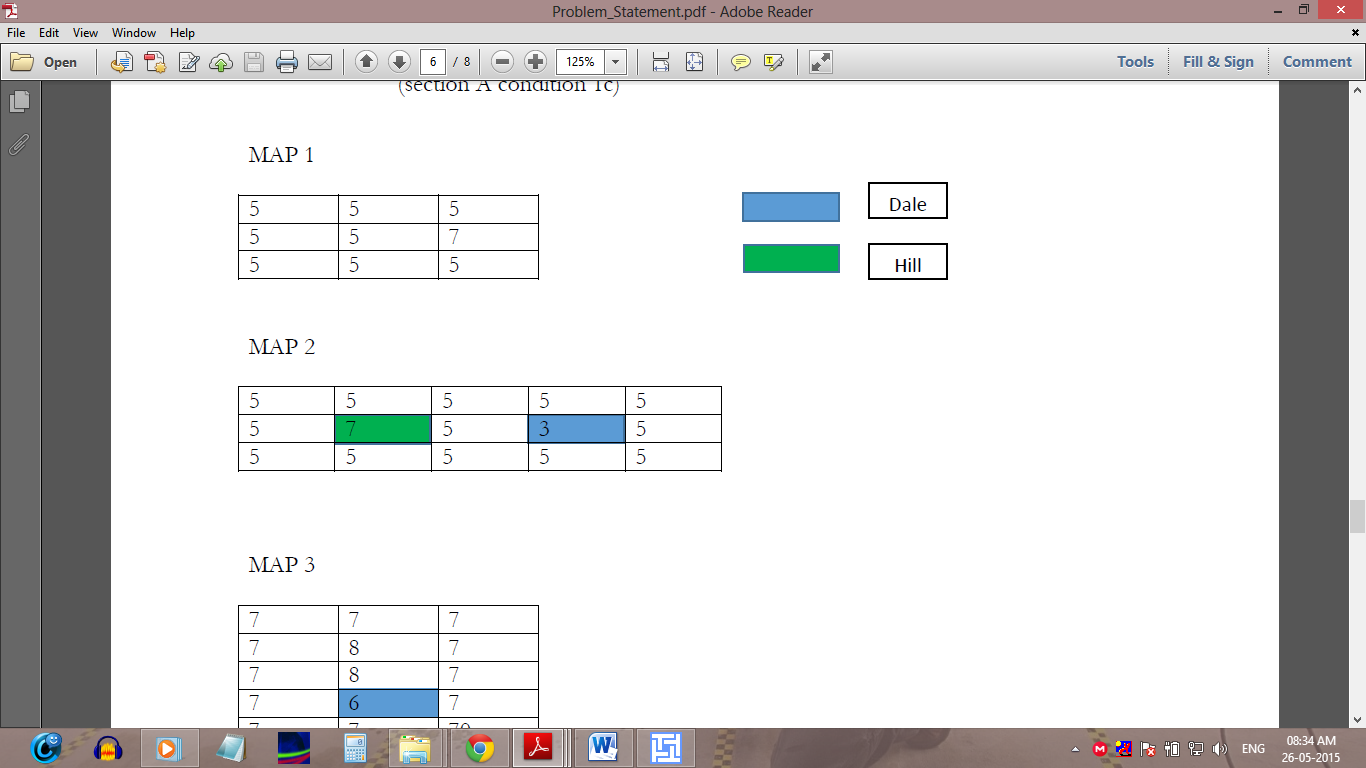
The CUDA runtime handles compiler translations from C-code to the GPU instruction set. It is an interface between the system and the GPU. The nvcc compiler invokes various specialized routines available in the runtime pertaining to code compilation and GPU-code optimizations.

# SOFTWARE WORKINGS

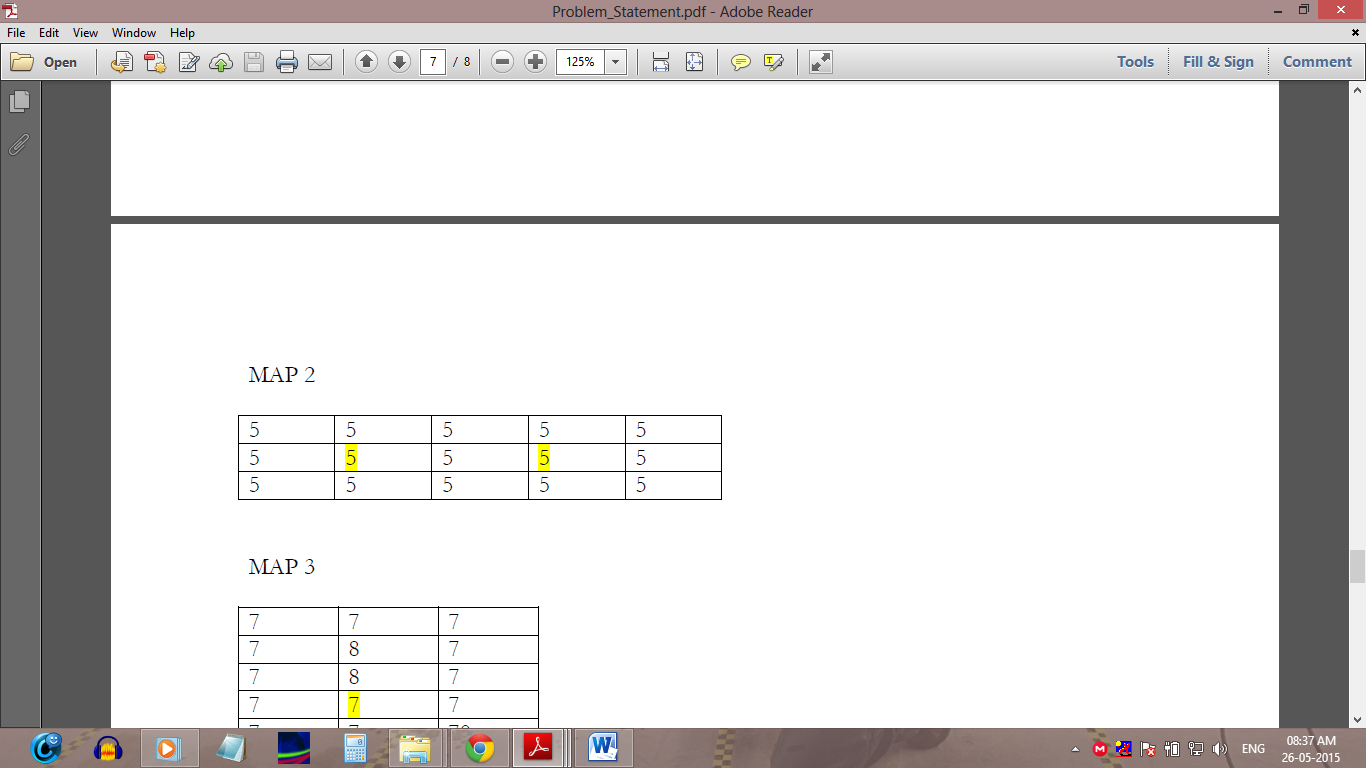
## Image Smoothening-

* For the given input set which contains N maps, find all possible Hills and Dales. Note that the input may contain multiple maps and each map may be of a different size.
* For all maps:
  + Replace all Hills with the mean value of the surrounding cells as defined above
  + Replace all Dales with median value of the surrounding cells as defined above
* For all maps, repeat Step 1 until either no Hills and Dales remain in the map or the number of iterations (including the first iteration in Step 1) reaches k. The value of k is defined in "input.h" as NUM\_ITERATIONS.
* At the end of each iteration a different map will be yielded and should be taken as the input for next iteration.

Illustrated below is the working of the image smoothening module-



ORIGNAL MAP



SMOOTHENED MAP

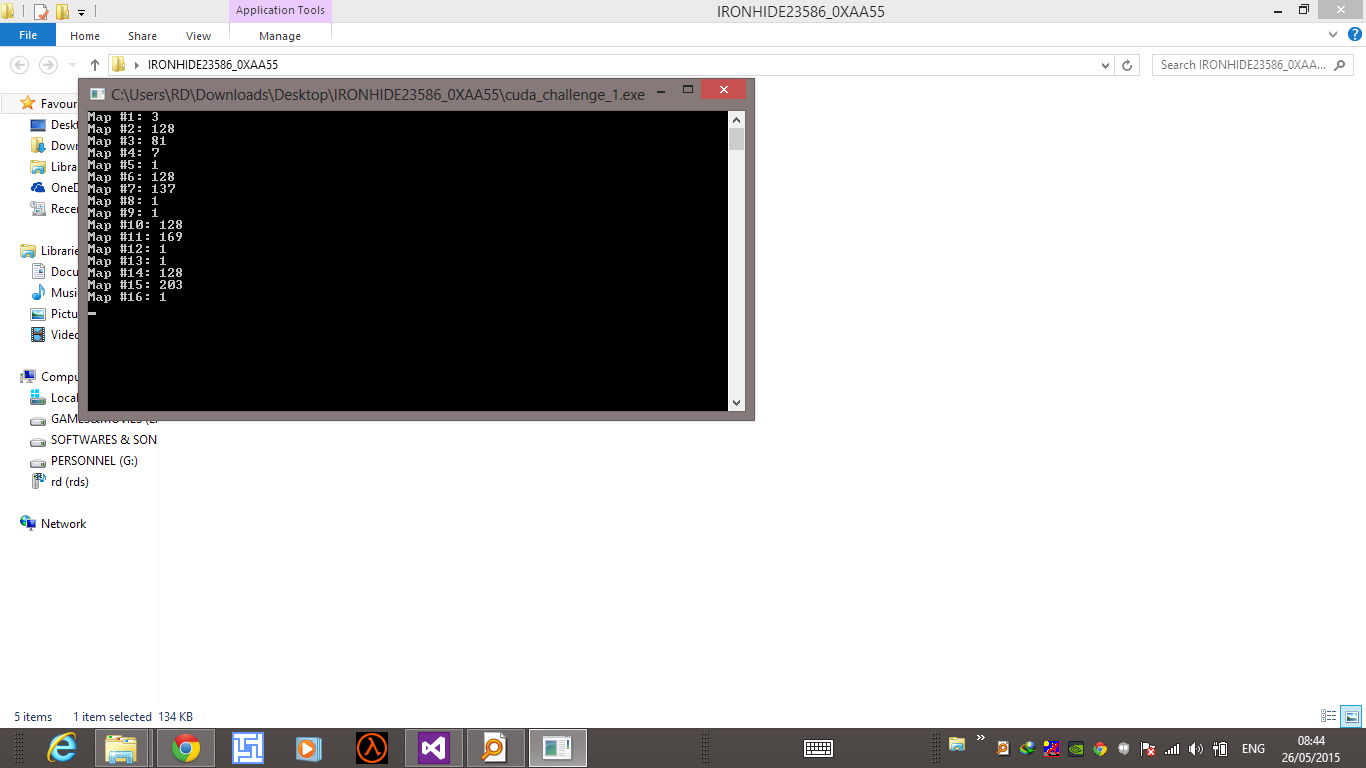
## Thresholding-

Convert Maps from Step 2 to binary Maps as follows:

* Compute the average value of a given Map by computing the average of all cells. This is the "Threshold", T. The average will be the sum of all cell values divided by total number of cells, rounded down to the smaller integer.
* For all cells in the map, assign labels as follows:
  1. If (Cell Value < T) Cell Value = 0;
  2. Else Cell Value = 1;

## Connected Component Counting-

* Count the number of connected components in all binary Maps from the previous step.
* The output will be the Map number followed by the corresponding number of connected components in that Map.
* We will take 8-connected objects (meaning that diagonally adjacent 1s are considered connected).



# CODE

#include <cuda.h>

#include <cuda\_runtime.h>

#include <cuda\_runtime\_api.h>

#include <stdio.h>

#include <input\_large.h>

#define indexFrom2D(i, j) ((i) \* cols + (j))

#define greatest(a, b) ((a > b) ? a : b)

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

#define MAP\_SAMPLING\_SIZE\_SIDE 3

#define MAP\_SAMPLING\_SIZE 9

#define LAST\_THREADBLOCK (effectiveBlockId == lastThreadBlockIndex)

#define THREADS\_PER\_BLOCK 1024

#define SERIAL\_SUM\_LIMIT 10

#define MAX\_NO\_OF\_THREADBLOCKS 65535

#define MAP\_ROOT h\_map[rootLinearIndex]

#define MAP\_RIGHT h\_map[rootRightIndex]

#define MAP\_DOWN h\_map[rootDownIndex]

#define MAP\_CORNER h\_map[rootCornerIndex]

\_\_host\_\_ \_\_device\_\_ unsigned int ceil\_h\_d(float f)

{

unsigned int tmp = (unsigned int) f;

if (f > tmp)

tmp++;

return tmp;

}

\_\_host\_\_ unsigned int nearestLesserSquare(unsigned int x)

{

unsigned int tmp = sqrt((float) x);

return tmp \* tmp;

}

\_\_host\_\_ unsigned int getReqThreadUnitsCountCCL(unsigned int rows, unsigned int cols)

{

return ceil\_h\_d((float) rows / 2) \* ceil\_h\_d((float) cols / 2);

}

\_\_device\_\_ unsigned int getRealRow(unsigned int cols, unsigned int effectiveblockIndex)

{

return (threadIdx.y - 1) + ceil\_h\_d((float) (effectiveblockIndex + 1) / (cols - 2));

}

\_\_device\_\_ unsigned int getRealCol(unsigned int cols, unsigned int effectiveblockIndex)

{

return threadIdx.x + (effectiveblockIndex % (cols - 2));

}

\_\_global\_\_ void processMapSampleHillDale(unsigned int \*d\_map, unsigned int rows, unsigned int cols, char \*d\_hillDaleRemaining, unsigned int threadBlockBatchIndex, unsigned int \*d\_aux\_map)

{

unsigned int effectiveBlockIndex = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int mapRow = threadIdx.y, mapCol = threadIdx.x;

unsigned int mapLinearIndex = mapRow \* 3 + mapCol;

unsigned int realRow = getRealRow(cols, effectiveBlockIndex), realCol = getRealCol(cols, effectiveBlockIndex);

unsigned int linearIndexFrom2D = indexFrom2D(realRow, realCol);

if (linearIndexFrom2D == 0)

d\_hillDaleRemaining[0] = 0;

\_\_shared\_\_ unsigned int mapContents[MAP\_SAMPLING\_SIZE\_SIDE][MAP\_SAMPLING\_SIZE\_SIDE];

\_\_shared\_\_ int cellDecider;

cellDecider = 0;

mapContents[mapRow][mapCol] = d\_map[linearIndexFrom2D];

\_\_syncthreads();

if (mapContents[mapRow][mapCol] < mapContents[1][1])

atomicAdd(&cellDecider, 1);

else if (mapContents[mapRow][mapCol] > mapContents[1][1])

atomicSub(&cellDecider, 1);

\_\_syncthreads();

if (cellDecider == 8) //HILL

{

//////////////PARALLEL AVERAGE

if (mapLinearIndex == 0)

{

mapContents[0][0] += mapContents[0][1];

d\_hillDaleRemaining[0] = 1;

}

else if (mapLinearIndex == 2)

mapContents[0][2] += mapContents[1][2];

else if (mapLinearIndex == 8)

mapContents[2][2] += mapContents[2][1];

else if (mapLinearIndex == 6)

mapContents[2][0] += mapContents[1][0];

\_\_syncthreads();

if (mapLinearIndex == 0)

mapContents[0][0] += mapContents[0][2];

else if (mapLinearIndex == 8)

mapContents[2][2] += mapContents[2][0];

\_\_syncthreads();

if (mapLinearIndex == 4)

{

(mapContents[0][0] += mapContents[2][2]) /= 8;

d\_aux\_map[linearIndexFrom2D] = mapContents[0][0];

}

}

else if (cellDecider == (-8)) //DALE

{

\_\_shared\_\_ unsigned int nos[8];

\_\_shared\_\_ char isSorted[2], maxPartitions;

if (mapLinearIndex == 4)

{

nos[4] = mapContents[2][2];

isSorted[0] = 0;

isSorted[1] = 0;

maxPartitions = 4;

d\_hillDaleRemaining[0] = 1;

}

else

nos[mapLinearIndex] = mapContents[mapRow][mapCol];

\_\_syncthreads();

unsigned int tmp, v1 = 2 \* mapLinearIndex, v2 = v1 + 1, v3 = v1 + 2;

while (!(isSorted[0] && isSorted[1]))

{

if (mapLinearIndex < 2)

isSorted[mapLinearIndex] = 1;

\_\_syncthreads();

if (mapLinearIndex < maxPartitions) //EVEN

{

if (nos[v1] > nos[v2])

{

tmp = nos[v2];

nos[v2] = nos[v1];

nos[v1] = tmp;

isSorted[0] = 0;

}

maxPartitions = 3;

}

\_\_syncthreads();

if (mapLinearIndex < maxPartitions) //ODD

{

if (nos[v2] > nos[v3])

{

tmp = nos[v2];

nos[v2] = nos[v3];

nos[v3] = tmp;

isSorted[1] = 0;

}

maxPartitions = 4;

}

\_\_syncthreads();

}

if (mapLinearIndex == 4)

{

(nos[3] += nos[4]) /= 2;

d\_aux\_map[linearIndexFrom2D] = nos[3];

}

}

}

\_\_global\_\_ void calculateAverage(unsigned int \*d\_map, unsigned int mapSize, unsigned int \*d\_average, unsigned int lastThreadBlockLastIndex, unsigned int lastThreadBlockIndex, char odd, unsigned int threadBlockBatchIndex)

{

unsigned tmp = threadIdx.x \* 2, effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int linearIndexFrom2D = (effectiveBlockId \* THREADS\_PER\_BLOCK \* 2) + tmp, incrLinearIndexFrom2D = linearIndexFrom2D + 1;

\_\_shared\_\_ unsigned int vals[THREADS\_PER\_BLOCK];

vals[threadIdx.x] = 0;

if (linearIndexFrom2D < mapSize)

{

vals[threadIdx.x] = d\_map[linearIndexFrom2D];

if (mapSize > incrLinearIndexFrom2D)

vals[threadIdx.x] += d\_map[incrLinearIndexFrom2D];

unsigned int adderIndex = tmp, incrAdderIndex = tmp + 1, i = 1;

unsigned int compareVal = LAST\_THREADBLOCK ? lastThreadBlockLastIndex : THREADS\_PER\_BLOCK;

\_\_syncthreads();

while (incrAdderIndex < compareVal)

{

vals[adderIndex] += vals[incrAdderIndex];

\_\_syncthreads();

i \*= 2;

adderIndex = i \* tmp;

incrAdderIndex = adderIndex + i;

}

if (threadIdx.x == 0)

{

d\_average[effectiveBlockId] = vals[0];

if (((LAST\_THREADBLOCK) && odd) && (lastThreadBlockLastIndex > 0))

d\_average[effectiveBlockId] += vals[lastThreadBlockLastIndex];

}

}

}

\_\_global\_\_ void threshold(unsigned int \*d\_map, unsigned int thresholdVal, unsigned int mapSize, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int linearIndexFrom2D = (effectiveBlockId \* THREADS\_PER\_BLOCK) + threadIdx.x;

if (linearIndexFrom2D < mapSize)

{

if (d\_map[linearIndexFrom2D] < thresholdVal)

d\_map[linearIndexFrom2D] = 0;

else

d\_map[linearIndexFrom2D] = 1;

}

}

\_\_global\_\_ void assignBlockNumbers(unsigned int \*d\_map, unsigned int rows, unsigned int cols, unsigned int blockMatrixRowLength, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int n = effectiveBlockId + 1, nFloor = n; //n -> Block ID to be assigned.

if ((n % blockMatrixRowLength) == 0)

nFloor--;

unsigned int rootLinearIndex = ((n - 1) / blockMatrixRowLength) \* cols + ((2 \* n) - 2 - ((nFloor / blockMatrixRowLength) \* ((cols % 2) > 0)));

unsigned int r = rootLinearIndex / cols, c = rootLinearIndex - r \* cols;

if (threadIdx.x < 2)

c += threadIdx.x;

else

{

r++;

c += threadIdx.x - 2;

}

unsigned int linearIndex = indexFrom2D(r, c);

if (((r < rows) && (c < cols)) && d\_map[linearIndex] == 1)

d\_map[linearIndex] = n;

}

\_\_global\_\_ void processMapSampleCCLZeroIndexHorizontal0(unsigned int \*d\_map, char \*d\_ifLesserFound, unsigned int rows, unsigned int cols, unsigned int blockMatrixRowLength, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int n = effectiveBlockId + 1, nFloor = n; //n -> Block ID to be assigned.

if ((n % blockMatrixRowLength) == 0)

nFloor--;

unsigned int rootLinearIndex = ((n - 1) / blockMatrixRowLength) \* cols + ((2 \* n) - 2 - ((nFloor / blockMatrixRowLength) \* ((cols % 2) > 0)));

unsigned int r = rootLinearIndex / cols, c = rootLinearIndex - r \* cols;

if (threadIdx.x < 2)

c += threadIdx.x;

else

{

r++;

c += threadIdx.x - 2;

}

unsigned int linearIndex = indexFrom2D(r, c);

\_\_shared\_\_ unsigned int sample[4], leastNum, leastCandidates[2];

if ((r < rows) && (c < cols))

sample[threadIdx.x] = d\_map[linearIndex];

else

sample[threadIdx.x] = 0;

\_\_syncthreads();

if (threadIdx.x < 2)

{

unsigned int tmp1 = threadIdx.x \* 2, tmp2 = tmp1 + 1;

if ((sample[tmp1] > 0) && (sample[tmp2] > 0))

{

if (sample[tmp1] < sample[tmp2])

{

d\_ifLesserFound[0] = 1;

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp2] < sample[tmp1])

{

d\_ifLesserFound[0] = 1;

leastCandidates[threadIdx.x] = sample[tmp2];

}

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp1] == 0)

leastCandidates[threadIdx.x] = sample[tmp2];

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

\_\_syncthreads();

if (threadIdx.x == 0)

{

if ((leastCandidates[0] > 0) && (leastCandidates[1] > 0))

{

if (leastCandidates[0] < leastCandidates[1])

{

d\_ifLesserFound[0] = 1;

leastNum = leastCandidates[0];

}

else if (leastCandidates[1] < leastCandidates[0])

{

d\_ifLesserFound[0] = 1;

leastNum = leastCandidates[1];

}

else

leastNum = leastCandidates[0];

}

else if (leastCandidates[0] == 0)

leastNum = leastCandidates[1];

else

leastNum = leastCandidates[0];

}

\_\_syncthreads();

if (((r < rows) && (c < cols)) && (d\_map[linearIndex] > 0))

d\_map[linearIndex] = leastNum;

}

\_\_global\_\_ void processMapSampleCCLOneIndexHorizontal1(unsigned int \*d\_map, char \*d\_ifLesserFound, unsigned int rows, unsigned int cols, unsigned int blockMatrixRowLength, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int n = effectiveBlockId + 1, nFloor = n;

if ((n % blockMatrixRowLength) == 0)

nFloor--;

unsigned int rootLinearIndex = (((n - 1) / blockMatrixRowLength) \* (cols)) + ((2 \* n) - 1 + ((nFloor / blockMatrixRowLength) \* ((cols % 2) > 0)));

unsigned int r = rootLinearIndex / cols, c = rootLinearIndex - r \* cols;

if (threadIdx.x < 2)

c += threadIdx.x;

else

{

r++;

c += threadIdx.x - 2;

}

unsigned int linearIndex = indexFrom2D(r, c);

\_\_shared\_\_ unsigned int sample[4], leastNum, leastCandidates[2];

if ((r < rows) && (c < cols))

sample[threadIdx.x] = d\_map[linearIndex];

else

sample[threadIdx.x] = 0;

\_\_syncthreads();

if (threadIdx.x < 2)

{

unsigned int tmp1 = threadIdx.x \* 2, tmp2 = tmp1 + 1;

if ((sample[tmp1] > 0) && (sample[tmp2] > 0))

{

if (sample[tmp1] < sample[tmp2])

{

d\_ifLesserFound[1] = 1;

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp2] < sample[tmp1])

{

d\_ifLesserFound[1] = 1;

leastCandidates[threadIdx.x] = sample[tmp2];

}

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp1] == 0)

leastCandidates[threadIdx.x] = sample[tmp2];

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

\_\_syncthreads();

if (threadIdx.x == 0)

{

if ((leastCandidates[0] > 0) && (leastCandidates[1] > 0))

{

if (leastCandidates[0] < leastCandidates[1])

{

d\_ifLesserFound[1] = 1;

leastNum = leastCandidates[0];

}

else if (leastCandidates[1] < leastCandidates[0])

{

d\_ifLesserFound[1] = 1;

leastNum = leastCandidates[1];

}

else

leastNum = leastCandidates[0];

}

else if (leastCandidates[0] == 0)

leastNum = leastCandidates[1];

else

leastNum = leastCandidates[0];

}

\_\_syncthreads();

if (((r < rows) && ((c > 0) && (c < cols))) && (d\_map[linearIndex] > 0))

d\_map[linearIndex] = leastNum;

}

\_\_global\_\_ void processMapSampleCCLZeroIndexVertical2(unsigned int \*d\_map, char \*d\_ifLesserFound, unsigned int rows, unsigned int cols, unsigned int blockMatrixRowLength, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int n = effectiveBlockId + 1, nFloor = n;

if ((n % blockMatrixRowLength) == 0)

nFloor--;

unsigned int rootLinearIndex = (((n - 1) / blockMatrixRowLength) \* (cols)) + (cols + (2 \* n) - 2 - ((nFloor / blockMatrixRowLength) \* ((cols % 2) > 0)));

unsigned int r = rootLinearIndex / cols, c = rootLinearIndex - r \* cols;

if (threadIdx.x < 2)

c += threadIdx.x;

else

{

r++;

c += threadIdx.x - 2;

}

unsigned int linearIndex = indexFrom2D(r, c);

\_\_shared\_\_ unsigned int sample[4], leastNum, leastCandidates[2];

if ((r < rows) && (c < cols))

sample[threadIdx.x] = d\_map[linearIndex];

else

sample[threadIdx.x] = 0;

\_\_syncthreads();

if (threadIdx.x < 2)

{

unsigned int tmp1 = threadIdx.x \* 2, tmp2 = tmp1 + 1;

if ((sample[tmp1] > 0) && (sample[tmp2] > 0))

{

if (sample[tmp1] < sample[tmp2])

{

d\_ifLesserFound[2] = 1;

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp2] < sample[tmp1])

{

d\_ifLesserFound[2] = 1;

leastCandidates[threadIdx.x] = sample[tmp2];

}

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp1] == 0)

leastCandidates[threadIdx.x] = sample[tmp2];

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

\_\_syncthreads();

if (threadIdx.x == 0)

{

if ((leastCandidates[0] > 0) && (leastCandidates[1] > 0))

{

if (leastCandidates[0] < leastCandidates[1])

{

d\_ifLesserFound[2] = 1;

leastNum = leastCandidates[0];

}

else if (leastCandidates[1] < leastCandidates[0])

{

d\_ifLesserFound[2] = 1;

leastNum = leastCandidates[1];

}

else

leastNum = leastCandidates[0];

}

else if (leastCandidates[0] == 0)

leastNum = leastCandidates[1];

else

leastNum = leastCandidates[0];

}

\_\_syncthreads();

if ((((r > 0) && (r < rows)) && (c < cols)) && (d\_map[linearIndex] > 0))

d\_map[linearIndex] = leastNum;

}

\_\_global\_\_ void processMapSampleCCLOneIndexVertical3(unsigned int \*d\_map, char \*d\_ifLesserFound, unsigned int rows, unsigned int cols, unsigned int blockMatrixRowLength, unsigned int threadBlockBatchIndex)

{

unsigned int effectiveBlockId = blockIdx.x + MAX\_NO\_OF\_THREADBLOCKS \* threadBlockBatchIndex;

unsigned int n = effectiveBlockId + 1, nFloor = n;

if ((n % blockMatrixRowLength) == 0)

nFloor--;

unsigned int rootLinearIndex = (((n - 1) / blockMatrixRowLength) \* (cols)) + (cols + (2 \* n) - 1 - ((nFloor / blockMatrixRowLength) \* ((cols % 2) > 0)));

unsigned int r = rootLinearIndex / cols, c = rootLinearIndex - r \* cols;

if (threadIdx.x < 2)

c += threadIdx.x;

else

{

r++;

c += threadIdx.x - 2;

}

unsigned int linearIndex = indexFrom2D(r, c);

\_\_shared\_\_ unsigned int sample[4], leastNum, leastCandidates[2];

if ((r < rows) && (c < cols))

sample[threadIdx.x] = d\_map[linearIndex];

else

sample[threadIdx.x] = 0;

\_\_syncthreads();

if (threadIdx.x < 2)

{

unsigned int tmp1 = threadIdx.x \* 2, tmp2 = tmp1 + 1;

if ((sample[tmp1] > 0) && (sample[tmp2] > 0))

{

if (sample[tmp1] < sample[tmp2])

{

d\_ifLesserFound[3] = 1;

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp2] < sample[tmp1])

{

d\_ifLesserFound[3] = 1;

leastCandidates[threadIdx.x] = sample[tmp2];

}

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

else if (sample[tmp1] == 0)

leastCandidates[threadIdx.x] = sample[tmp2];

else

leastCandidates[threadIdx.x] = sample[tmp1];

}

\_\_syncthreads();

if (threadIdx.x == 0)

{

if ((leastCandidates[0] > 0) && (leastCandidates[1] > 0))

{

if (leastCandidates[0] < leastCandidates[1])

{

d\_ifLesserFound[3] = 1;

leastNum = leastCandidates[0];

}

else if (leastCandidates[1] < leastCandidates[0])

{

d\_ifLesserFound[3] = 1;

leastNum = leastCandidates[1];

}

else

leastNum = leastCandidates[0];

}

else if (leastCandidates[0] == 0)

leastNum = leastCandidates[1];

else

leastNum = leastCandidates[0];

}

\_\_syncthreads();

if ((((r > 0) && (r < rows)) && ((c > 0) && (c < cols))) && (d\_map[linearIndex] > 0))

d\_map[linearIndex] = leastNum;

}

int main()

{

dim3 dimMap(MAP\_SAMPLING\_SIZE\_SIDE, MAP\_SAMPLING\_SIZE\_SIDE);

unsigned int \*h\_input = get\_input();

unsigned int NO\_OF\_MAPS = h\_input[0];

unsigned int rootLinearIndex, rootRightIndex, rootDownIndex, rootCornerIndex, leastNum1, leastNum2, leastNum, finalRowIndex, finalColIndex;

unsigned int rows = h\_input[1], cols = h\_input[2];

unsigned int currMapSize = rows \* cols, r, c;

unsigned int inputReadIndex = 3, currMapSizeBytes = currMapSize \* sizeof(unsigned int), currMapSizeChar = currMapSize \* sizeof(char); //Begin reading map

unsigned int \*d\_map; //Variable to hold map read on GPU.

unsigned int \*h\_map;

unsigned int \*h\_buff, topIndex, k, l, j;

unsigned int \*h\_average;

char tmp;

unsigned int mapCount = 0, mapCountStop = NO\_OF\_MAPS - 1, iterations;

unsigned int \*h\_input\_tmp = &h\_input[inputReadIndex], reqThreadUnits1, reqThreadUnits2, reqThreadUnits3, reqThreadUnits4;

char \*d\_hillDaleRemaining, h\_hillDaleRemaining;

unsigned int \*d\_average1, \*d\_average2, finalAverage, n, reqThreadUnits2FullBlockCount, maxMapDimensionLength, i;

unsigned int threadBlockBatchCount, extraThreadBlockCount;

cudaMalloc((void \*\*) &d\_hillDaleRemaining, sizeof(char));

float threadBlockBatchCountFloat;

unsigned int \*d\_aux\_map;

unsigned int lastTBLastIndex;

unsigned int lastThreadBlockElementCount;

char odd;

unsigned int reqThreadUnits2Bytes;

unsigned int reqThreadUnits3Bytes;

char \*d\_ifLesserFound;

unsigned int reqThreadUnits3FullBlockCount;

char h\_ifLesserFound[4];

while (true)

{

cudaMalloc((void \*\*) &d\_map, currMapSizeBytes);

cudaMemcpy(d\_map, h\_input\_tmp, currMapSizeBytes, cudaMemcpyHostToDevice);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MAP PROCESSING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

//------------------------------------HILL & DALE-------------------------------//

reqThreadUnits1 = (rows - 2) \* (cols - 2);

iterations = 0;

h\_hillDaleRemaining = 1;

threadBlockBatchCountFloat = (float) reqThreadUnits1 / MAX\_NO\_OF\_THREADBLOCKS;

cudaMalloc((void \*\*) &d\_aux\_map, currMapSizeBytes);

cudaMemcpy(d\_aux\_map, d\_map, currMapSizeBytes, cudaMemcpyDeviceToDevice);

if (threadBlockBatchCountFloat <= 1)

{

while ((h\_hillDaleRemaining) && (iterations < NUM\_ITERATIONS))

{

processMapSampleHillDale<<<reqThreadUnits1, dimMap>>>(d\_map, rows, cols, d\_hillDaleRemaining, 0, d\_aux\_map);

cudaMemcpy(d\_map, d\_aux\_map, currMapSizeBytes, cudaMemcpyDeviceToDevice);

iterations++;

cudaMemcpy(&h\_hillDaleRemaining, d\_hillDaleRemaining, sizeof(h\_hillDaleRemaining), cudaMemcpyDeviceToHost);

}

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits1 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

while ((h\_hillDaleRemaining) && (iterations < NUM\_ITERATIONS))

{

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

{

processMapSampleHillDale<<<MAX\_NO\_OF\_THREADBLOCKS, dimMap>>>(d\_map, rows, cols, d\_hillDaleRemaining, i, d\_aux\_map);

}

processMapSampleHillDale<<<extraThreadBlockCount, dimMap>>>(d\_map, rows, cols, d\_hillDaleRemaining, threadBlockBatchCount, d\_aux\_map);

cudaMemcpy(d\_map, d\_aux\_map, currMapSizeBytes, cudaMemcpyDeviceToDevice);

iterations++;

cudaMemcpy(&h\_hillDaleRemaining, d\_hillDaleRemaining, sizeof(h\_hillDaleRemaining), cudaMemcpyDeviceToHost);

}

}

cudaMemcpy(d\_map, d\_aux\_map, currMapSizeBytes, cudaMemcpyDeviceToDevice);

cudaFree(d\_aux\_map);

//------------------------------------HILL & DALE-------------------------------//

//------------------------------------AVERAGE FINDING-------------------------------//

reqThreadUnits2 = ceil\_h\_d((float) ceil\_h\_d((float) currMapSize / 2) / THREADS\_PER\_BLOCK);

reqThreadUnits2FullBlockCount = (reqThreadUnits2 - 1) \* THREADS\_PER\_BLOCK;

cudaMalloc((void \*\*) &d\_average1, reqThreadUnits2 \* sizeof(unsigned int));

lastTBLastIndex = ceil\_h\_d((float) currMapSize / 2 - reqThreadUnits2FullBlockCount) - 1;

lastThreadBlockElementCount = currMapSize - reqThreadUnits2FullBlockCount \* 2;

odd = ~(lastThreadBlockElementCount && (!(lastThreadBlockElementCount & (lastThreadBlockElementCount - 1)))); //To check if lastThreadBlockElementCount is a power of 2

threadBlockBatchCountFloat = (float) reqThreadUnits2 / MAX\_NO\_OF\_THREADBLOCKS;

if ((threadBlockBatchCountFloat <= 1))

{

calculateAverage<<<reqThreadUnits2, THREADS\_PER\_BLOCK>>>(d\_map, currMapSize, d\_average1, lastTBLastIndex, (reqThreadUnits2 - 1), odd, 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits1 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

calculateAverage<<<MAX\_NO\_OF\_THREADBLOCKS, THREADS\_PER\_BLOCK>>>(d\_map, currMapSize, d\_average1, lastTBLastIndex, (reqThreadUnits2 - 1), odd, i);

}

calculateAverage<<<extraThreadBlockCount, THREADS\_PER\_BLOCK>>>(d\_map, currMapSize, d\_average1, lastTBLastIndex, (reqThreadUnits2 - 1), odd, threadBlockBatchCount);

}

maxMapDimensionLength = greatest(rows, cols);

if (reqThreadUnits2 < SERIAL\_SUM\_LIMIT)

{

reqThreadUnits2Bytes = reqThreadUnits2 \* sizeof(unsigned int);

h\_average = (unsigned int \*)malloc(reqThreadUnits2Bytes);

cudaMemcpy(h\_average, d\_average1, reqThreadUnits2Bytes, cudaMemcpyDeviceToHost);

finalAverage = 0;

for (n = 0; n < reqThreadUnits2; n++)

{

finalAverage += h\_average[n];

}

finalAverage /= currMapSize;

}

else

{

finalAverage = 0;

reqThreadUnits3 = ceil\_h\_d((float) ceil\_h\_d((float) reqThreadUnits2 / 2) / THREADS\_PER\_BLOCK);

reqThreadUnits3Bytes = reqThreadUnits3 \* sizeof(unsigned int);

reqThreadUnits3FullBlockCount = (reqThreadUnits3 - 1) \* THREADS\_PER\_BLOCK;

lastTBLastIndex = ceil\_h\_d((float) reqThreadUnits2 / 2 - reqThreadUnits3FullBlockCount) - 1;

lastThreadBlockElementCount = reqThreadUnits2 - reqThreadUnits3FullBlockCount \* 2;

odd = ~(lastThreadBlockElementCount && (!(lastThreadBlockElementCount & (lastThreadBlockElementCount - 1))));

cudaMalloc((void \*\*) &d\_average2, reqThreadUnits3Bytes);

calculateAverage<<<reqThreadUnits3, THREADS\_PER\_BLOCK>>>(d\_average1, reqThreadUnits2, d\_average2, lastTBLastIndex, (reqThreadUnits3 - 1), odd, 0);

h\_average = (unsigned int \*)malloc(reqThreadUnits3Bytes);

cudaMemcpy(h\_average, d\_average2, reqThreadUnits3Bytes, cudaMemcpyDeviceToHost);

for (n = 0; n < reqThreadUnits3; n++)

{

finalAverage += h\_average[n];

}

finalAverage /= currMapSize;

}

//------------------------------------AVERAGE FINDING-------------------------------//

//------------------------------------THRESHOLDING-------------------------------//

reqThreadUnits3 = ceil\_h\_d((float) currMapSize / THREADS\_PER\_BLOCK);

threadBlockBatchCountFloat = (float) reqThreadUnits3 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

threshold<<<reqThreadUnits3, THREADS\_PER\_BLOCK>>>(d\_map, finalAverage, currMapSize, 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits3 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

threshold<<<MAX\_NO\_OF\_THREADBLOCKS, THREADS\_PER\_BLOCK>>>(d\_map, finalAverage, currMapSize, i);

}

threshold<<<extraThreadBlockCount, THREADS\_PER\_BLOCK>>>(d\_map, finalAverage, currMapSize, threadBlockBatchCount);

}

//------------------------------------THRESHOLDING-------------------------------//

//------------------------------------CONNECTED COMPONENT LABELLING-------------------------------//

reqThreadUnits1 = getReqThreadUnitsCountCCL(rows, cols);

reqThreadUnits2 = getReqThreadUnitsCountCCL(rows, (cols - 1));

reqThreadUnits3 = getReqThreadUnitsCountCCL((rows - 1), cols);

reqThreadUnits4 = getReqThreadUnitsCountCCL((rows - 1), (cols - 1));

//########################################################################################################//

threadBlockBatchCountFloat = (float) reqThreadUnits1 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

assignBlockNumbers<<<reqThreadUnits1, 4>>>(d\_map, rows, cols, ceil\_h\_d((float) cols / 2), 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits1 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

assignBlockNumbers<<<MAX\_NO\_OF\_THREADBLOCKS, 4>>>(d\_map, rows, cols, ceil\_h\_d((float) cols / 2), i);

}

assignBlockNumbers<<<extraThreadBlockCount, 4>>>(d\_map, rows, cols, ceil\_h\_d((float) cols / 2), threadBlockBatchCount);

}

//########################################################################################################//

finalRowIndex = rows - 1;

finalColIndex = cols - 1;

h\_map = (unsigned int \*)malloc(currMapSizeBytes);

cudaMemcpy(h\_map, d\_map, currMapSizeBytes, cudaMemcpyDeviceToHost);

for (r = 0; r < finalRowIndex; r++)

{

for (c = 0; c < finalColIndex; c++)

{

rootLinearIndex = indexFrom2D(r, c);

rootRightIndex = indexFrom2D(r, c + 1);

rootDownIndex = indexFrom2D(r + 1, c);

rootCornerIndex = indexFrom2D(r + 1, c + 1);

if ((MAP\_ROOT > 0) && (MAP\_RIGHT > 0))

leastNum1 = least(MAP\_ROOT, MAP\_RIGHT);

else if (MAP\_ROOT == 0)

leastNum1 = MAP\_RIGHT;

else

leastNum1 = MAP\_ROOT;

if ((MAP\_DOWN > 0) && (MAP\_CORNER > 0))

leastNum2 = least(MAP\_DOWN, MAP\_CORNER);

else if (MAP\_DOWN == 0)

leastNum2 = MAP\_CORNER;

else

leastNum2 = MAP\_DOWN;

if ((leastNum1 > 0) && (leastNum2 > 0))

leastNum = least(leastNum1, leastNum2);

else if (leastNum1 == 0)

leastNum = leastNum2;

else

leastNum = leastNum1;

if (leastNum > 0)

{

if (MAP\_ROOT > 0)

MAP\_ROOT = leastNum;

if (MAP\_RIGHT > 0)

MAP\_RIGHT = leastNum;

if (MAP\_DOWN > 0)

MAP\_DOWN = leastNum;

if (MAP\_CORNER > 0)

MAP\_CORNER = leastNum;

}

}

}

cudaMemcpy(d\_map, h\_map, currMapSizeBytes, cudaMemcpyHostToDevice);

free(h\_map);

h\_ifLesserFound[0] = 1;

cudaMalloc((void \*\*) &d\_ifLesserFound, 4 \* sizeof(char));

while (h\_ifLesserFound[0] | h\_ifLesserFound[1] | h\_ifLesserFound[2] | h\_ifLesserFound[3])

{

cudaMemset(d\_ifLesserFound, 0, 4 \* sizeof(char));

threadBlockBatchCountFloat = (float) reqThreadUnits2 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

processMapSampleCCLOneIndexHorizontal1<<<reqThreadUnits2, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits2 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

processMapSampleCCLOneIndexHorizontal1<<<MAX\_NO\_OF\_THREADBLOCKS, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), i);

}

processMapSampleCCLOneIndexHorizontal1<<<extraThreadBlockCount, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), threadBlockBatchCount);

}

threadBlockBatchCountFloat = (float) reqThreadUnits3 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

processMapSampleCCLZeroIndexVertical2<<<reqThreadUnits3, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits3 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

processMapSampleCCLZeroIndexVertical2<<<MAX\_NO\_OF\_THREADBLOCKS, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), i);

}

processMapSampleCCLZeroIndexVertical2<<<extraThreadBlockCount, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), threadBlockBatchCount);

}

threadBlockBatchCountFloat = (float) reqThreadUnits4 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

processMapSampleCCLOneIndexVertical3<<<reqThreadUnits4, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits4 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

processMapSampleCCLOneIndexVertical3<<<MAX\_NO\_OF\_THREADBLOCKS, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), i);

}

processMapSampleCCLOneIndexVertical3<<<extraThreadBlockCount, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) (cols - 1) / 2), threadBlockBatchCount);

}

threadBlockBatchCountFloat = (float) reqThreadUnits1 / MAX\_NO\_OF\_THREADBLOCKS;

if (threadBlockBatchCountFloat <= 1)

{

processMapSampleCCLZeroIndexHorizontal0<<<reqThreadUnits1, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), 0);

}

else

{

threadBlockBatchCount = (unsigned int) threadBlockBatchCountFloat;

extraThreadBlockCount = reqThreadUnits1 - threadBlockBatchCount \* MAX\_NO\_OF\_THREADBLOCKS;

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

{

processMapSampleCCLZeroIndexHorizontal0<<<MAX\_NO\_OF\_THREADBLOCKS, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), i);

}

processMapSampleCCLZeroIndexHorizontal0<<<extraThreadBlockCount, 4>>>(d\_map, d\_ifLesserFound, rows, cols, ceil\_h\_d((float) cols / 2), threadBlockBatchCount);

}

cudaMemcpy(h\_ifLesserFound, d\_ifLesserFound, 4 \* sizeof(char), cudaMemcpyDeviceToHost);

}

//------------------------------------CONNECTED COMPONENT LABELLING-------------------------------//

//------------------------------------CONNECTED COMPONENT COUNTING-------------------------------//

h\_map = (unsigned int \*)malloc(currMapSizeBytes);

cudaMemcpy(h\_map, d\_map, currMapSizeBytes, cudaMemcpyDeviceToHost);

cudaFree(d\_map);

h\_buff = (unsigned int \*)malloc(currMapSizeBytes);

memset(h\_buff, 0, currMapSizeBytes);

topIndex = 0;

for (k = 0; k < currMapSize; k++)

{

tmp = 0;

if (h\_map[k] > 0)

{

for (l = 0; l <= topIndex; l++)

{

if (h\_buff[l] == h\_map[k])

{

tmp = 1;

l = topIndex + 1;

}

}

if (!tmp)

h\_buff[topIndex++] = h\_map[k];

}

}

free(h\_map);

free(h\_buff);

//------------------------------------CONNECTED COMPONENT COUNTING-------------------------------//

printf("Map #%u: %u\n", mapCount + 1, topIndex);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MAP PROCESSING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

if (mapCount == mapCountStop)

break;

inputReadIndex += currMapSize;

rows = h\_input[inputReadIndex++];

cols = h\_input[inputReadIndex++];

h\_input\_tmp = &h\_input[inputReadIndex];

currMapSize = rows \* cols;

currMapSizeBytes = currMapSize \* sizeof(unsigned int);

currMapSizeChar = currMapSize \* sizeof(char);

mapCount++;

}

}

# Development Softwares Required

* **VISUAL STUDIO 2012**

Visual Studio Software is a comprehensive software development IDE from Microsoft which allows for software development for any windows based platform in various languages. This software in conjunction with the CUDA toolkit for Visual Studio was used to develop the software.

* **CUDA Libraries**

These libraries implement various GPU hardware level functionalities in the form of special directives. Functionalites include thread synchronization, management of deveice and host memory and other crucial ones for which a software implementation would either be impossible or too computationally complex.

# CONCLUSION

The objective was to develop a parallel implementation of the Connected Component Labelling algorithm on the GPU.

Judging by the different functionalities and utilities of the software as discussed above, it is safe to say that all the requirements have been fulfilled.

Furthermore, the software modules may be re-used as it is, in other projects eliciting the use of CCL algorithms.

# BIBLIOGRAPHY

* [www.stackoverflow.com](http://www.stackoverflow.com)
* [www.codeproject.com](http://www.codeproject.com)
* [www.dreamincode.com](http://www.dreamincode.com)
* [www.nvidia.com](http://www.nvidia.com)
* [www.udacity.com](http://www.udacity.com)
* [www.wikipedia.org](http://www.wikipedia.org)

# EXAMINER’S REMARKS