



# Parallel Implementation of Connected Component Labelling in NVIDIA® CUDA (Compute Unified Device Architecture)

---

CSE Main-Project 8th  
Semester

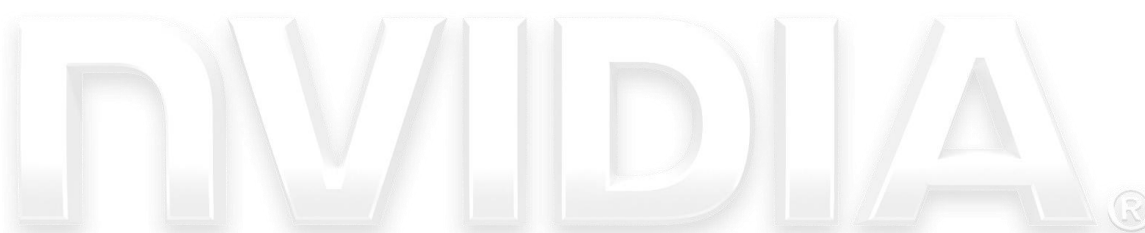
---

Souham Biswas,  
B.Tech. Computer Science & Engineering  
Roll No. – 25

---

## Table of Contents

<b>INTRODUCTION</b> .....	2
ACKNOWLEDGEMENT .....	3
<b>CERTIFICATE</b> .....	4
SOFTWARE PRE-REQUISITES .....	5
SOFTWARE OVERVIEW .....	6
MAIN FUNCTIONAL OUTLINE .....	8
FEATURES .....	10
TECHNOLOGIES AND SOFTWARES USED .....	10
SOFTWARE WORKINGS .....	13
Image Smoothing- .....	13
Thresholding- .....	14
Connected Component Counting- .....	14
CODE .....	15
Development Softwares Required .....	34
CONCLUSION .....	35
BIBLIOGRAPHY .....	36
<b>EXAMINER'S REMARKS</b> .....	37

A large, light gray, 3D-style NVIDIA logo is centered on the page. It is a watermark and not a primary element of the document's design.

# 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

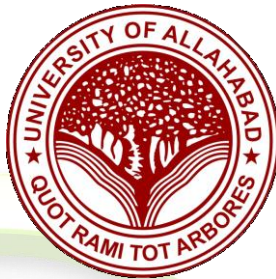
I would like to thank Prof. R.R. Tewari for his invaluable guidance provided throughout the project development.

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.

NVIDIA®

# 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: \_\_\_\_\_

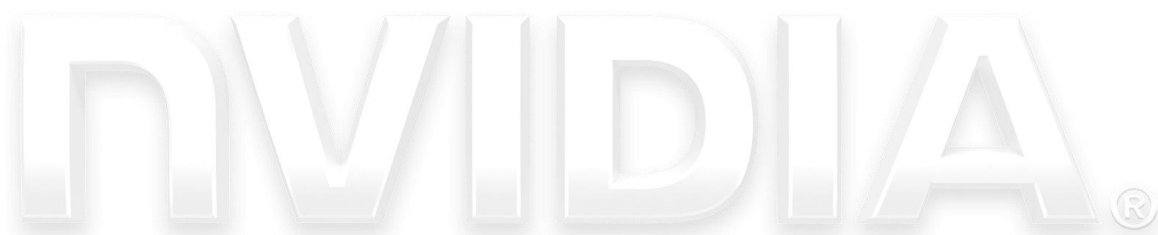
## **SOFTWARE PRE-REQUISITES**

### ➤ **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.

A large, light green, semi-transparent NVIDIA logo is centered in the background of the page. It features the word "NVIDIA" in a bold, sans-serif font, with a registered trademark symbol (®) to the right. The logo is slightly tilted and has a soft glow effect.



## 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.

- i) A cell is a HILL if all its 8 adjacent cells have a lower altitude.
- ii) A cell is a DALE if all its 8 adjacent cells have a higher altitude.
- iii) 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).

5	5	5	5	5
5	7	5	3	5
5	5	5	5	5

### 2. 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	1	0	0	0	1	1
0	1	0	1	1	0	0
1	0	0	1	1	0	1
1	0	0	0	0	0	1
0	1	0	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0

### 3. 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

$$Mean(X) = \frac{A + B + C + D + E + F + G + H}{8}$$

### 4. 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-

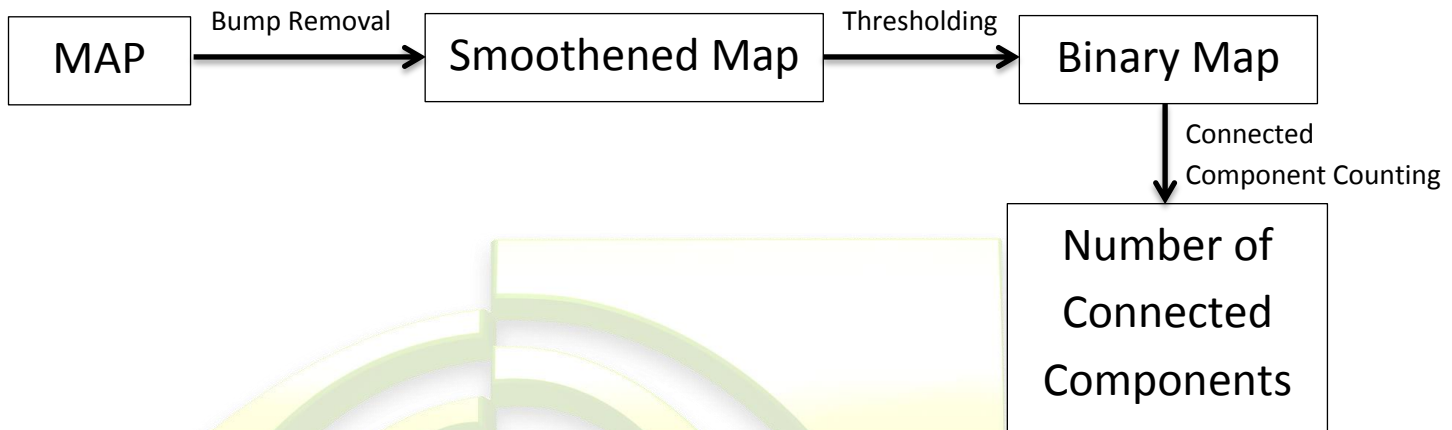
Sort the surrounding cells and,

$$Median(X) = \frac{Value\ at\ 4th\ location + Value\ at\ 5th\ location}{2}$$



## MAIN FUNCTIONAL OUTLINE

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



### ➤ 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 smoothed 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 smoothed 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 connected 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 one.

### ▪ **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 explained 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.

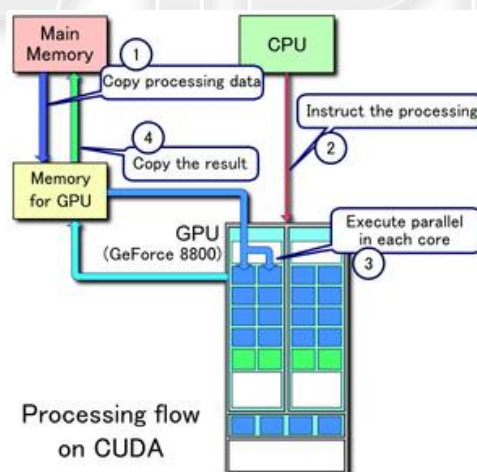
The language “CUDA-C” is a minor modification of the much popular C-language. Program constructs known as ‘kernels’ have been introduced which form the building blocks of any CUDA code. The kernels are basically threads which are to be executed in parallel on the different execution units (known as Streaming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

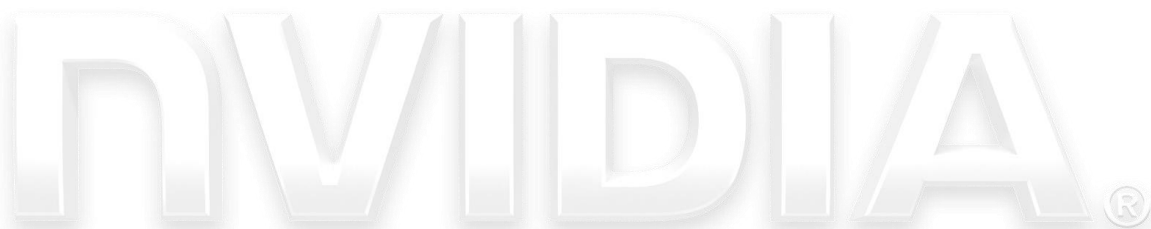
Illustrated below, is an example of a CUDA processing flow -



- 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 Smoothing-

- 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 smoothing module-

5	5	5	5	5
5	7	5	3	5
5	5	5	5	5

ORIGINAL MAP

5	5	5	5	5
5	5	5	5	5
5	5	5	5	5

SMOOTHENED MAP



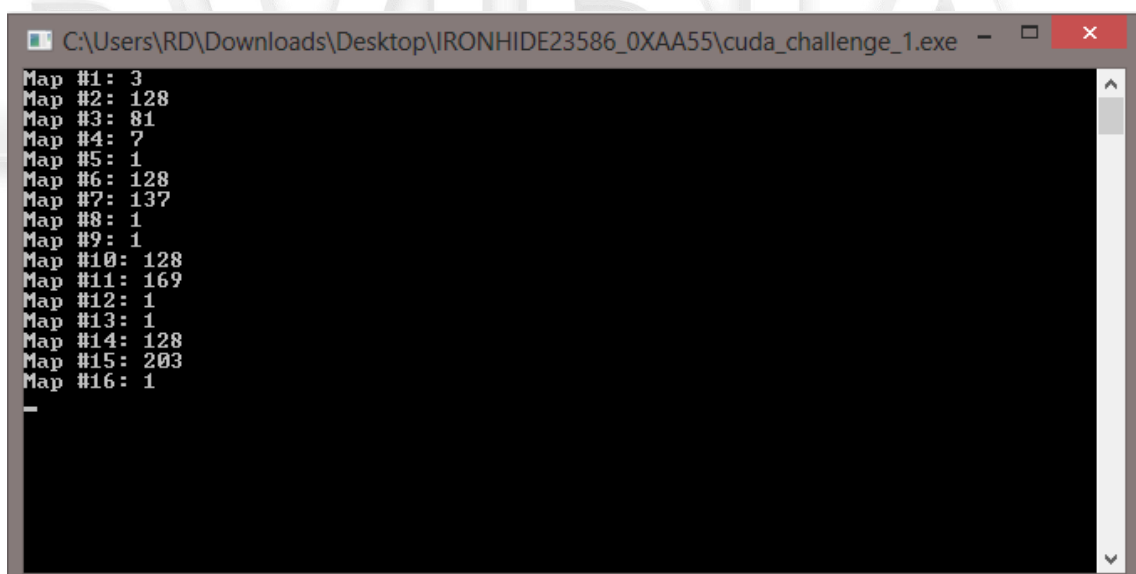
## Thresholding-

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:
  - a) If  $(\text{Cell Value} < T)$  Cell Value = 0;
  - b) 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).



```
C:\Users\RD\Downloads\Desktop\IRONHIDE23586_0XAA55\cuda_challenge_1.exe
Map #1: 3
Map #2: 128
Map #3: 81
Map #4: 7
Map #5: 1
Map #6: 128
Map #7: 137
Map #8: 1
Map #9: 1
Map #10: 128
Map #11: 169
Map #12: 1
Map #13: 1
Map #14: 128
Map #15: 203
Map #16: 1
```

## 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++;  
    }  
}
```



**NVIDIA®**

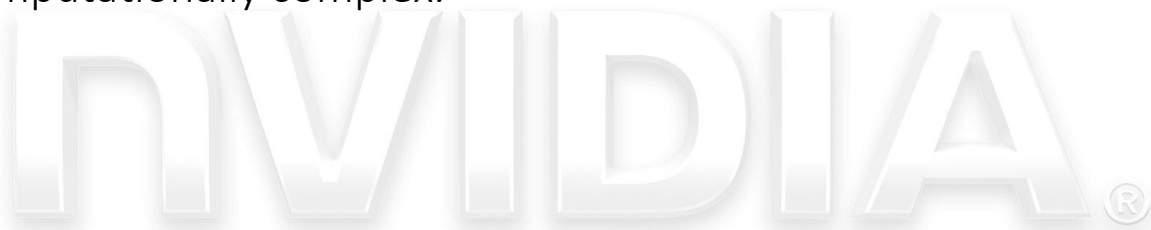
## Development Softwares Required

- **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.

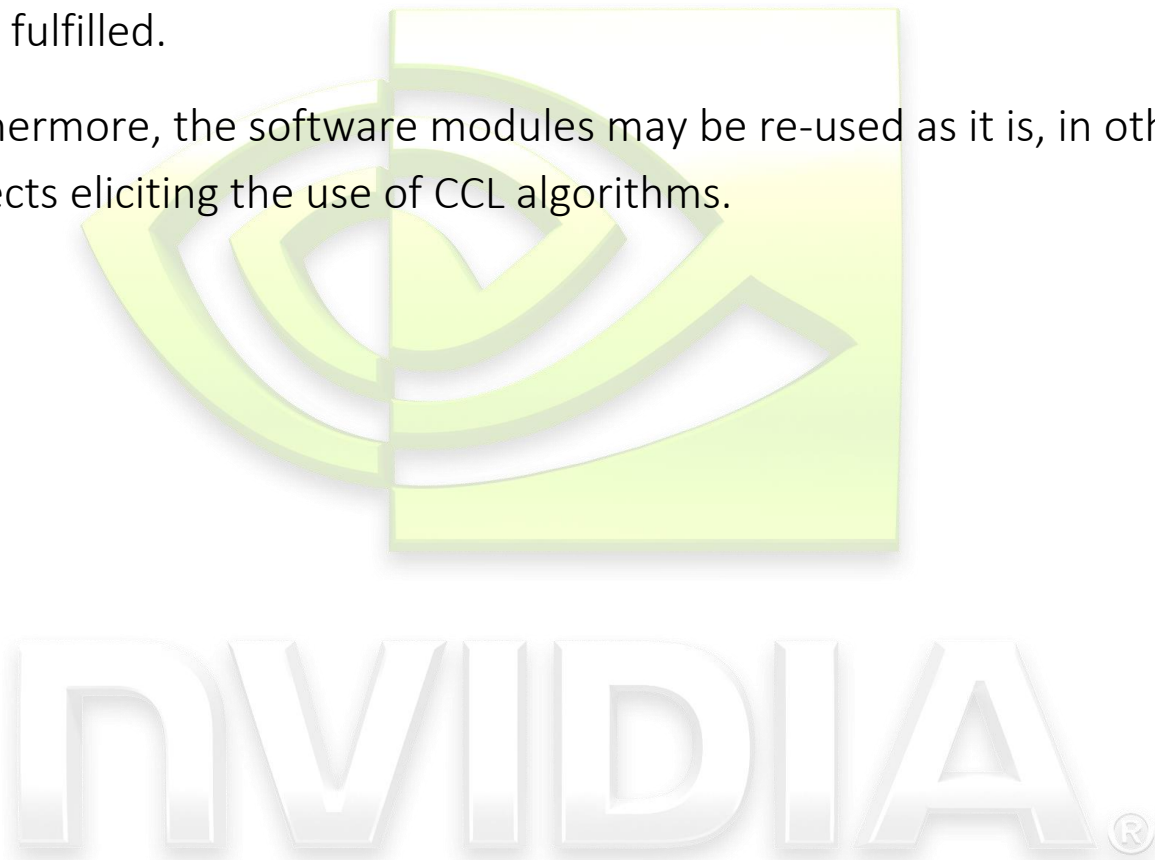
A large, semi-transparent, 3D-style NVIDIA logo is centered in the background of the text area. It features the word "NVIDIA" in a bold, sans-serif font, with a registered trademark symbol (®) to the right. The letters have a slight shadow and a gradient, giving them a three-dimensional appearance.

## 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



**NVIDIA®**