Implementation of Algorithms using CUDA, and their Performance Comparison

Anuja B. S.	*1NH10CS019*
Chaitra J. R.	*1NH10CS022*
Kusuma Chandrika K.	*1NH10CS047*
Nitin Kaveriappa U. M.	*1NH10CS061*

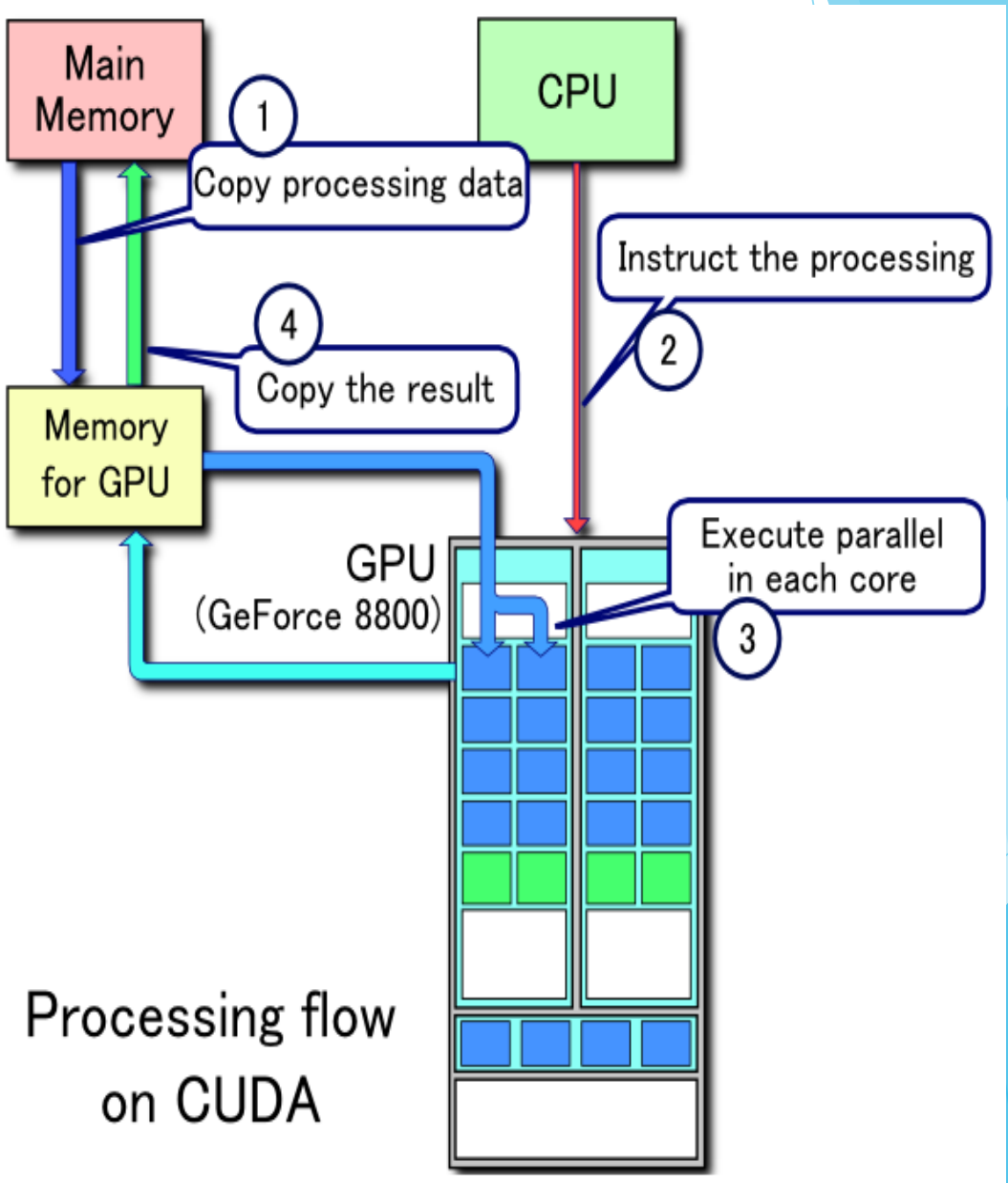
Objective

- This project is focused on exploiting GPU (Graphics Processing Unit) for general purpose computing based on CUDA (Compute Unified Device Architecture).
- It aims at utilization of GPU cores through direct access of GPU computing using CUDA that enables straightforward implementation of parallel algorithms.
- Diamond Square, Quickhull and Smith Waterman algorithms were implemented in CUDA C and C language and the algorithms were compared on performance and efficiency.

Introduction

- **Compute Unified Device Architecture (CUDA)** is a parallel computing platform
- Programming model created by NVIDIA
- Implemented by the graphics processing units (GPUs) that they produce.

Processing Flow on CUDA



General Keywords and Functions

- Header file: The header file used for CUDA is #include<cuda.h>
- _global_:

  The __global__ qualifier declares a function as being a kernel.

  Executed on the device, callable from the host,

  __global__ functions must have void return type.

  Ex: __global__ void subhtKernel( );
- cudaMalloc():

  A single address space is used for the host and all the devices.

  Syntax: cudaMalloc(void ** pointer, size_t nbytes)

  Ex: cudaMalloc((void**) &d_j, sizeof(int));

- **cudaMemcpy():**

  Memory is typically allocated using **cudaMalloc()** and freed using **cudaFree().**

  Data transfer between host memory and device memory are done through this function.

  Syntax: cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

  Ex: cudaMemcpy(d_j, &j, sizeof(int), cudaMemcpyHostToDevice);

  cudaMemcpy(d_j, &j, sizeof(int), cudaMemcpyDeviceToHost);

  Syntax: cudaFree(void* pointer)

  Ex: cudaFree(d_j);

- **Function call(kernel):**

  CUDA threads that execute that kernel for a given kernel call is specified using a new

  <<<...>>> *execution configuration*.

  Syntax: Function_name<<< no.of blocks, no.of threads/blocks>>>

  Ex: subhtKernel<<<1, j>>>( );

  Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable.

- **cudaDeviceSynchronize():**

  Blocks until the device has completed all preceding requested tasks. CudaDeviceSynchronize() returns an error if one of the preceding tasks has failed.

- **__shared__:**

  The __shared__ qualifier, declares a variable that:

  - Resides in the shared memory space of a thread block,
  - Has the lifetime of the block,
  - Is only accessible from all the threads within the block.

  When declaring a variable in shared memory as an external array such as

  Ex: extern __shared__ float shared[];

- **threadIdx:**

  threadIdx is a 3-component vector

  threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index.

  Ex: int tid = threadIdx.x;(1-d)

- **__syncthreads():**

  If the threads belong to the same block, in which case they should use **__syncthreads()** and share data through shared memory.

CUDA Function

```
__global__ void subhtKernel(int *limit)
{
	extern __shared__ kernel_shared_type sdata[];
	unsigned int tid = threadIdx.x;
	if (tid < *limit)
	{
		__syncthreads();
	}
}

points subht(int j)
{
	int *d_j;
	cudaMalloc((void**) &d_j, sizeof(int));
	cudaMemcpy(d_j, &j, sizeof(int), cudaMemcpyHostToDevice);
	subhtKernel<<<1, j>>>(d_j,);
	cudaDeviceSynchronize();
	cudaMemcpy(&j, d_j, sizeof(int), cudaMemcpyDeviceToHost);
	cudaFree(d_j);
}
```

Non - CUDA Function

```
points subht(int j)
{
	double h[200];
	for(i=0;i<j;i++)
	{
		//find the point with max distance from the baseline
	}
}
```

CUDA AND IT'S DOMAINS

- *Graphical Rendering*

Volume rendering using 2-3 swap image compositing

GPU accelerated volume rendering on an interactive light field display

Diamond Square Algorithm

- *Geometric Computation*

Geometric problems on 3D meshes

Parallel Convex Hull Algorithm

Quickhull Algorithm

- *Genomics*

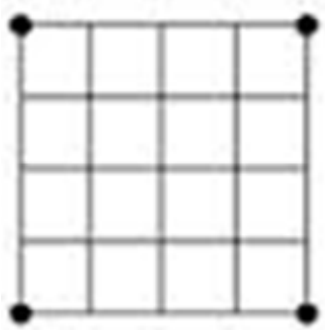
Smith Waterman Algorithm

Introduction to Diamond Square Algorithm

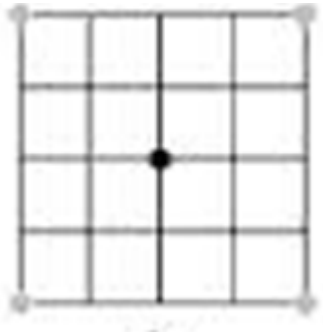
- The **diamond-square algorithm** is a method for generating height-maps for computer graphics.

Diamond Square Algorithm

- Step 1: Create a 2D array. The size of the array is determined by the amount of iterations (i.e. 1 iteration-3x3 array, 2 iterations-5x5 array etc).
- Step 2: Assign a start value to the four corners of the map.



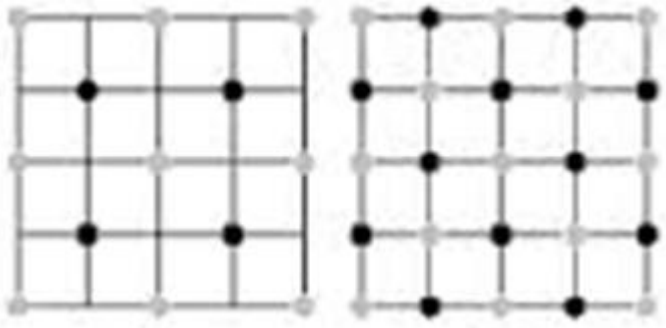
- Step 3: Call the Diamond() method.



Diamond Square Algorithm

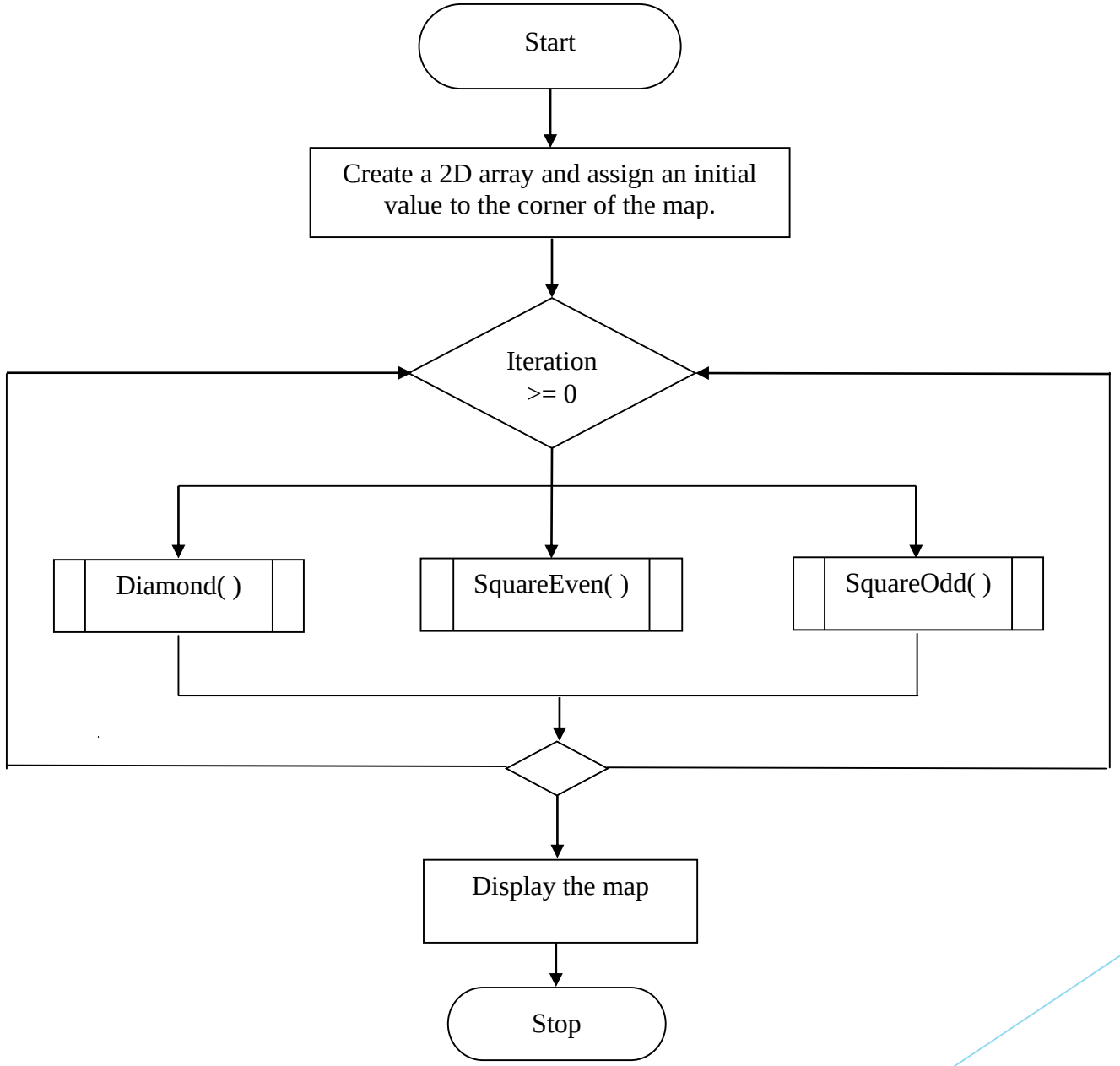
- Step 4: Call the SquareEven() method.
- Step 5: Call the SquareOdd() method.

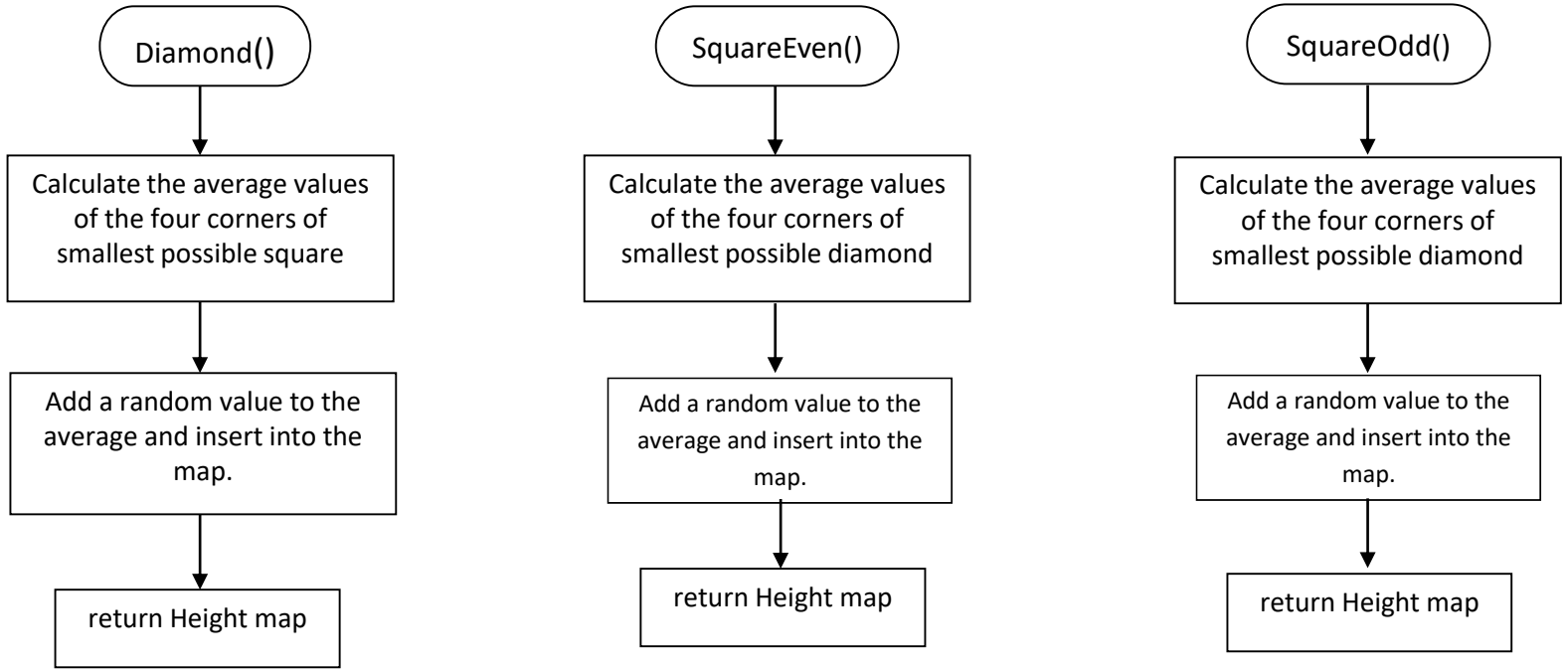
- Step 6: Go back to step 3 and repeat the steps until iterations >= 0.



- Step 7: Display the map.

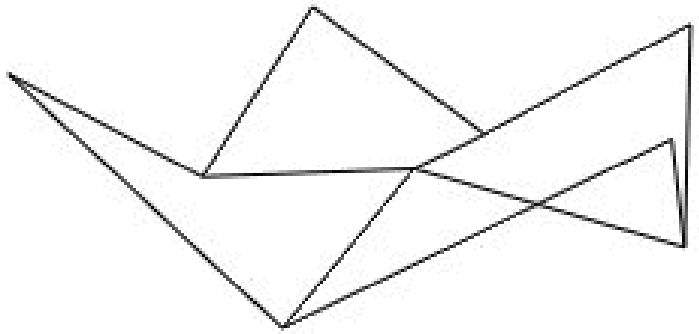
Flowchart for Diamond Square Algorithm

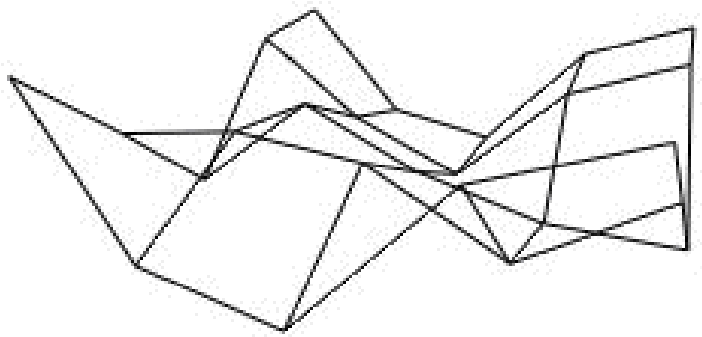



Diamond()
Calculate the average values of the four corners of smallest possible square
Add a random value to the average and insert into the map.
return Height map
SquareEven()
Calculate the average values of the four corners of smallest possible diamond
Add a random value to the average and insert into the map.
return Height map
SquareOdd()
Calculate the average values of the four corners of smallest possible diamond
Add a random value to the average and insert into the map.
return Height map

Implementation

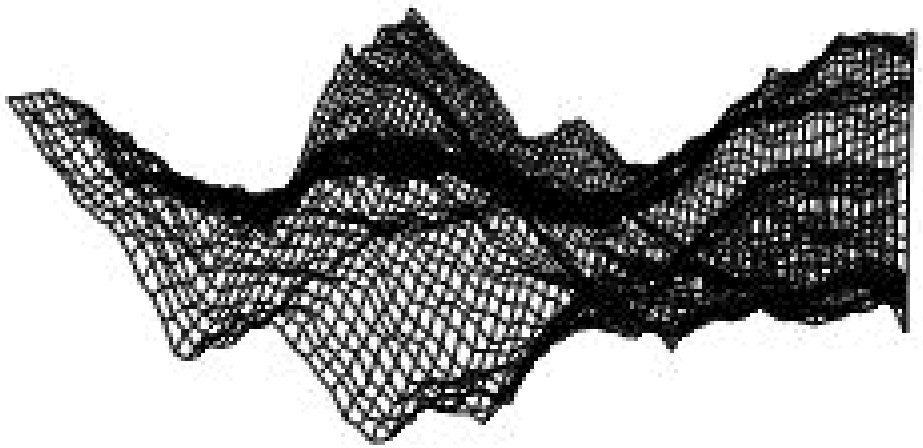
- After the first pass if we were to connect the nine points , we might get a wireframe surface which looks like this:



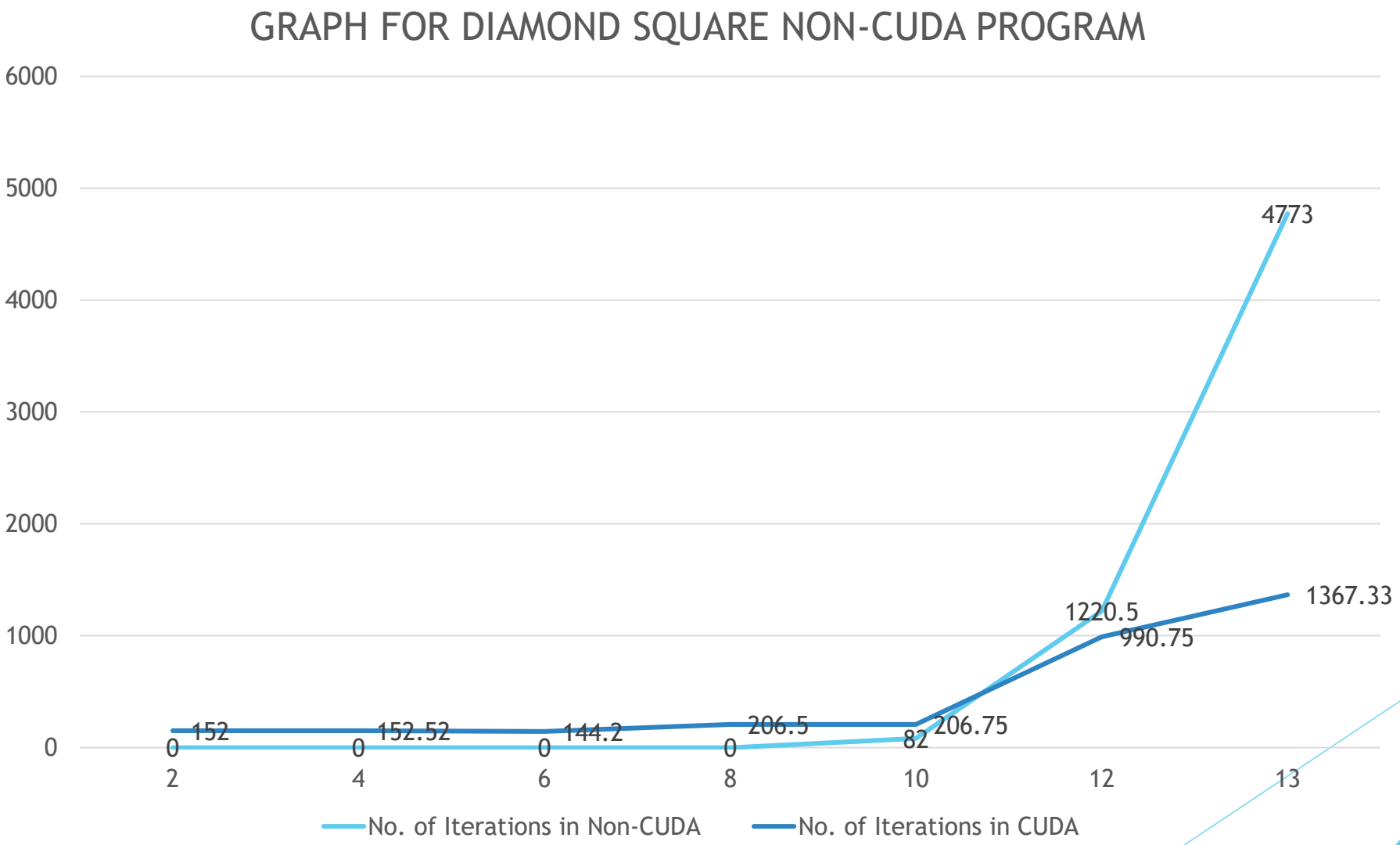


- When we perform the second pass with 12 diamond centres ,we need to calculate 12 new values . 25 elements of the array have been arranged and the wireframe would look like above:

- Had a larger array been allocated we could add more detail in each pass and the surface would like this



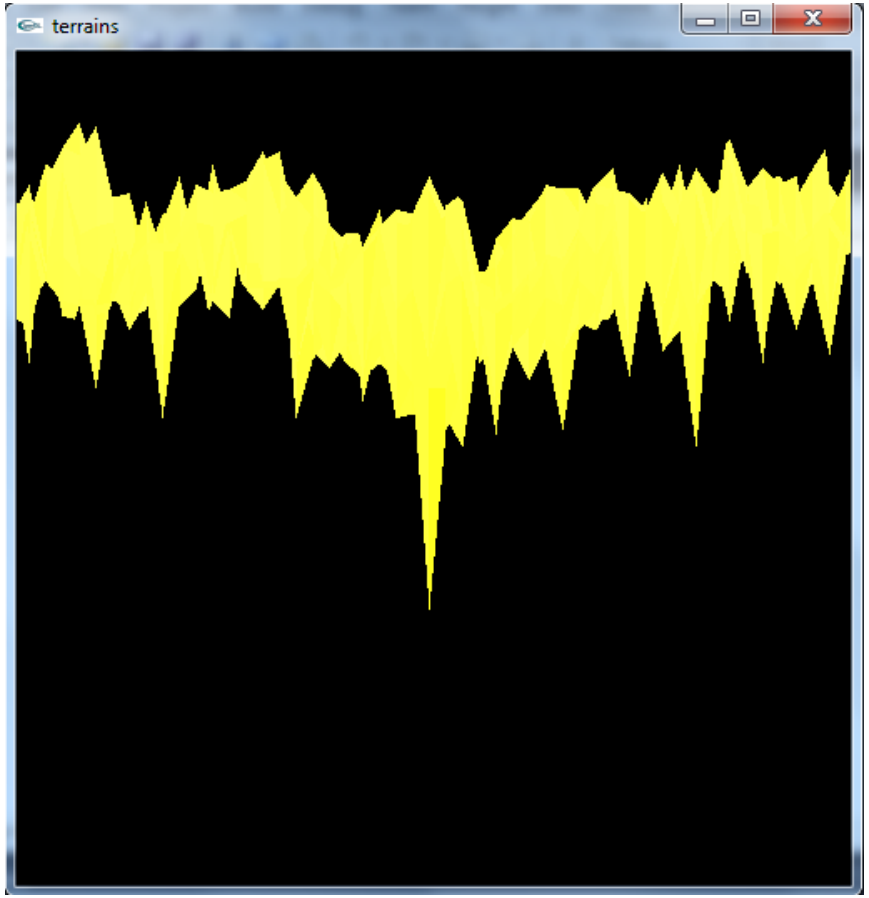
Non-CUDA Performance Graph

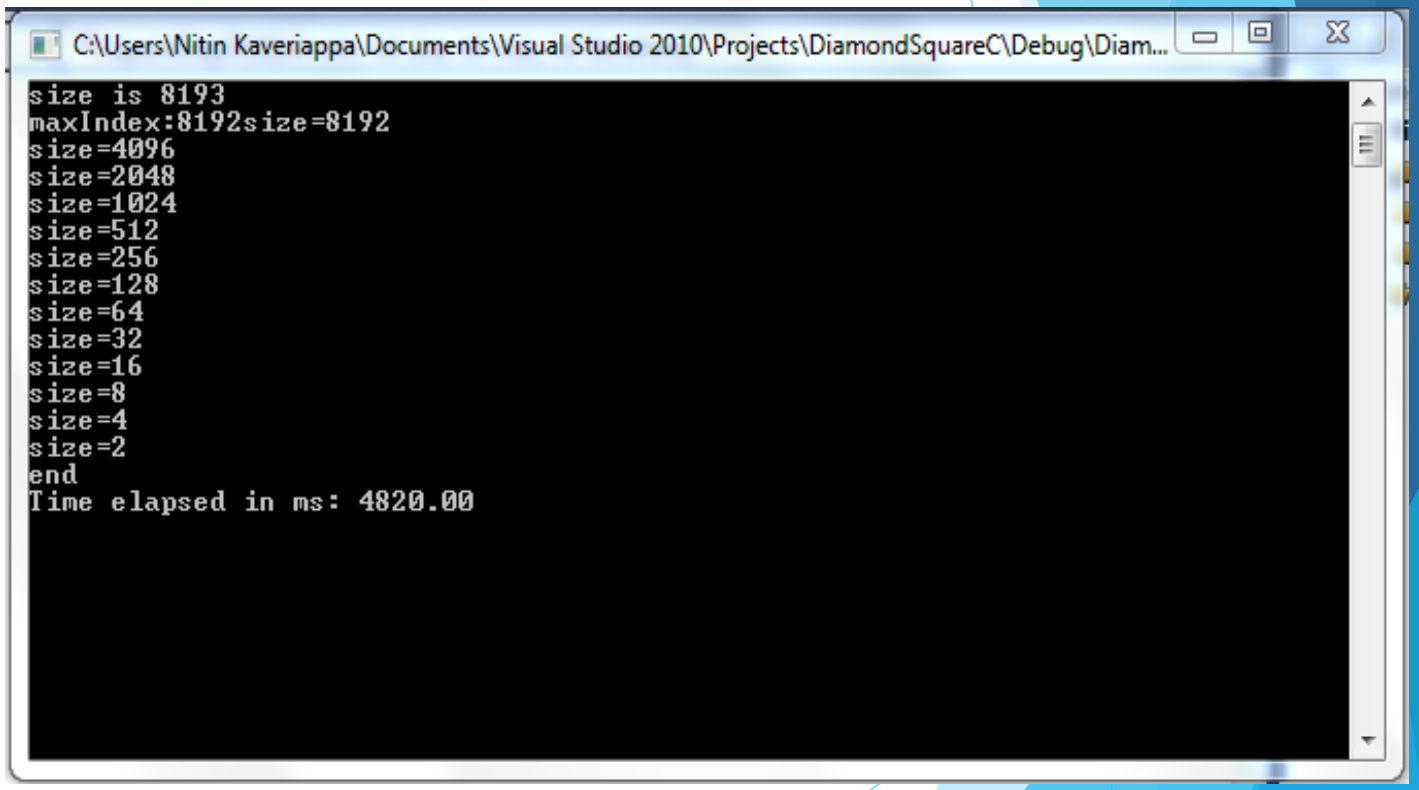


Performance Comparison

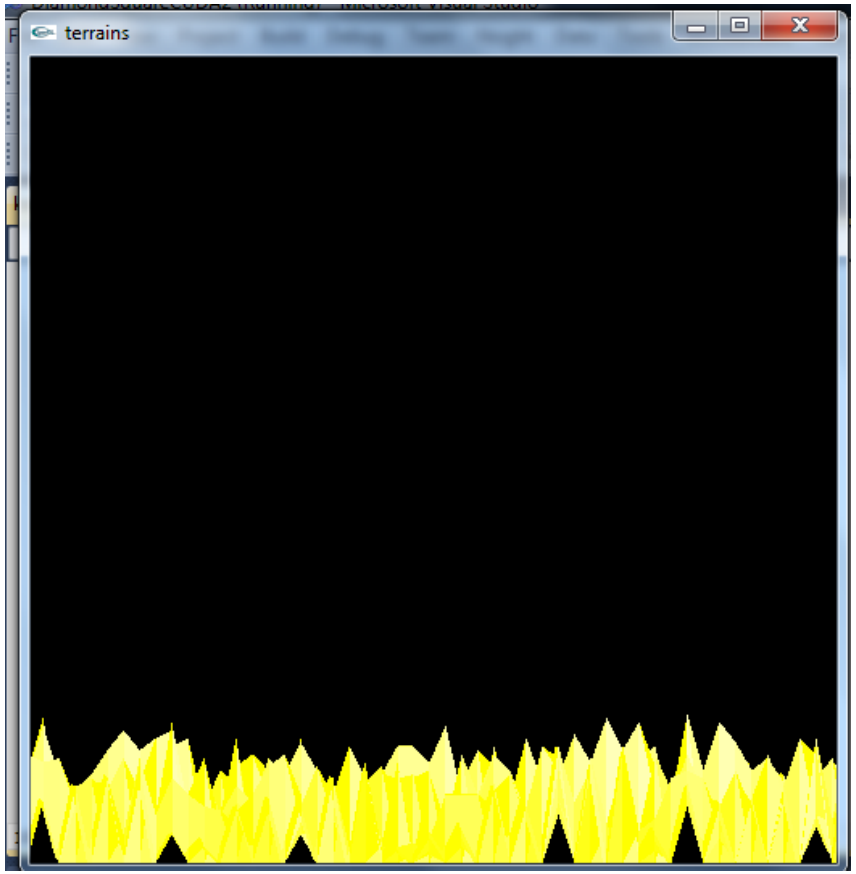
No. of iterations	Non CUDA execution time(ms)	CUDA execution time(ms)
2	0.00	152.00
4	0.00	152.25
6	0.00	144.25
8	0.00	156.00
10	82.00	206.75
12	1220.50	990.75
13	4779.00	1367.33

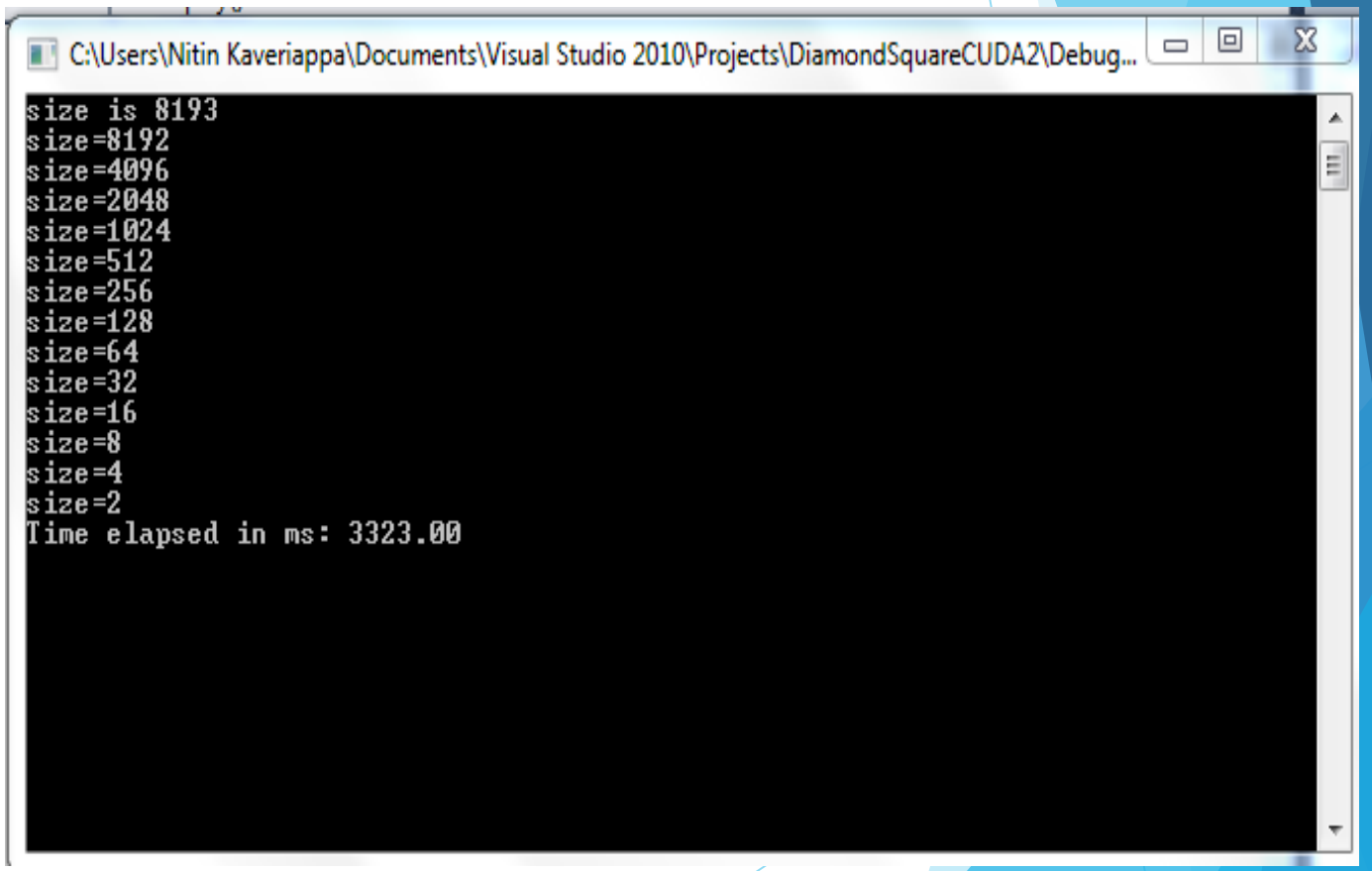
Non-CUDA Output





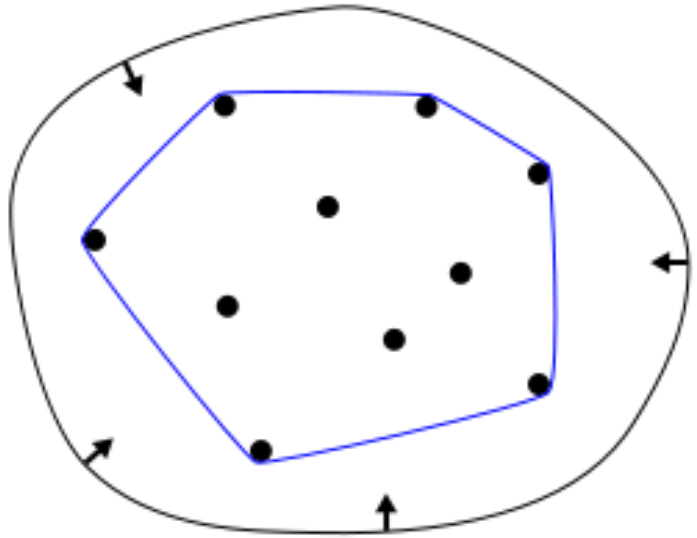
CUDA Output


terrains


C:\Users\Nitin Kaveriappa\Documents\Visual Studio 2010\Projects\DiamondSquareCUDA2\Debug...
size is 8193
size=8192
size=4096
size=2048
size=1024
size=512
size=256
size=128
size=64
size=32
size=16
size=8
size=4
size=2
Time elapsed in ms: 3323.00

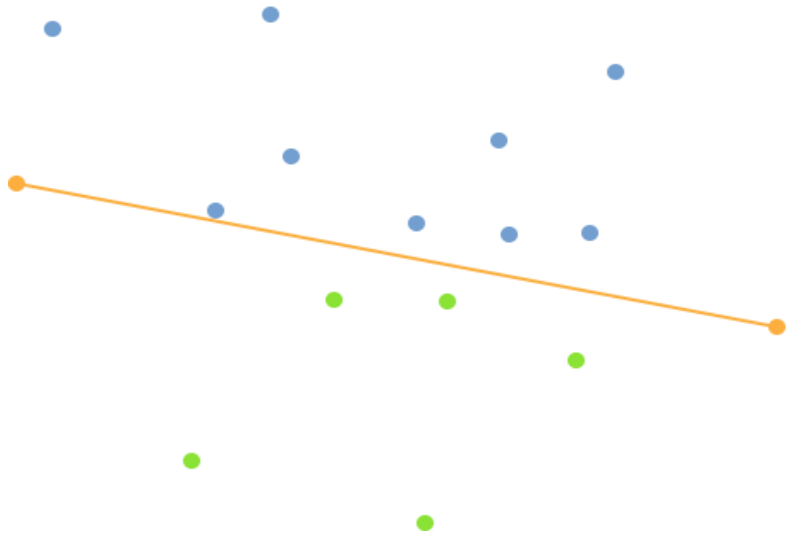
Introduction to QuickHull Algorithm

- **QuickHull** is a method of computing the convex hull of a finite set of points in the plane.
- It uses a divide and conquer approach.

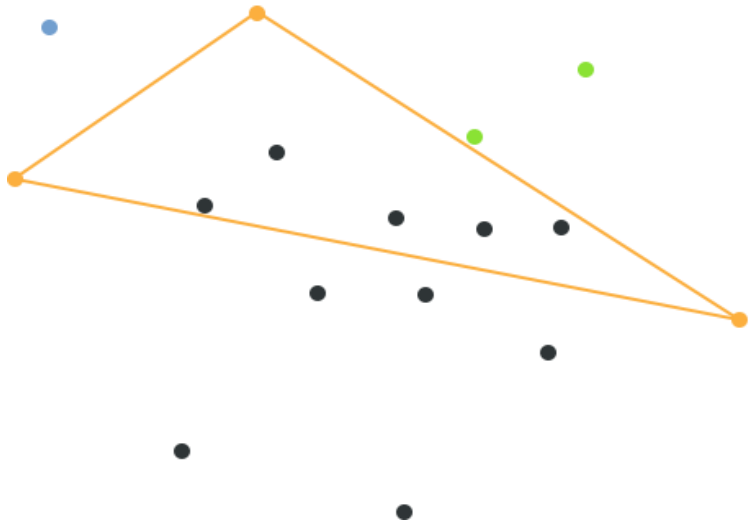


QuickHull Algorithm

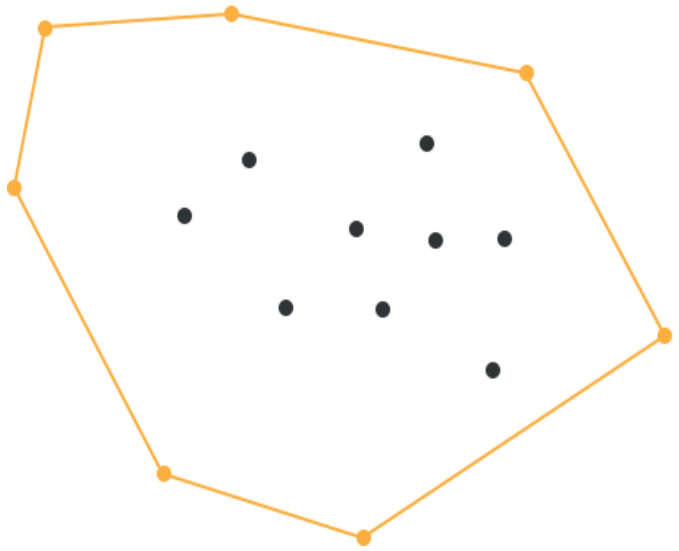
- Step 1: Points consists of x and y coordinates.
- Step 2: Find two points furthest away from each other and this is baseline PQ.
- Step 3: This PQ line divides points into 2 subsets S_1 and S_2.



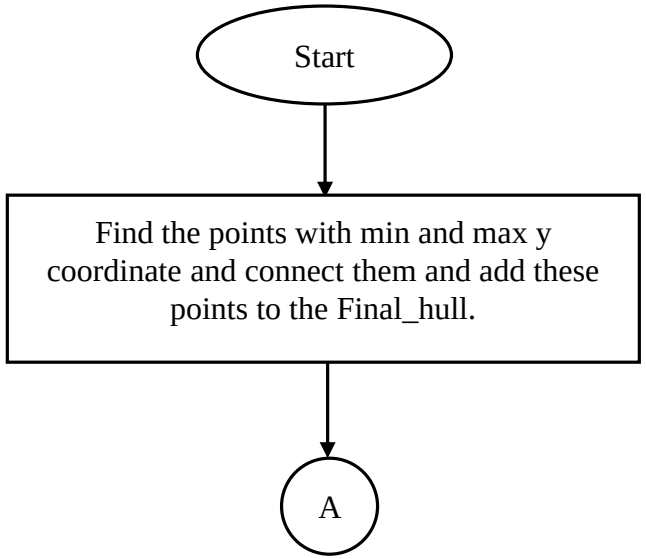
- Step 4: From line segment PQ find the point with the maximum distance from the line.
  Let it be ‘A’.
- A triangle is formed and the points inside the triangle is ignored.

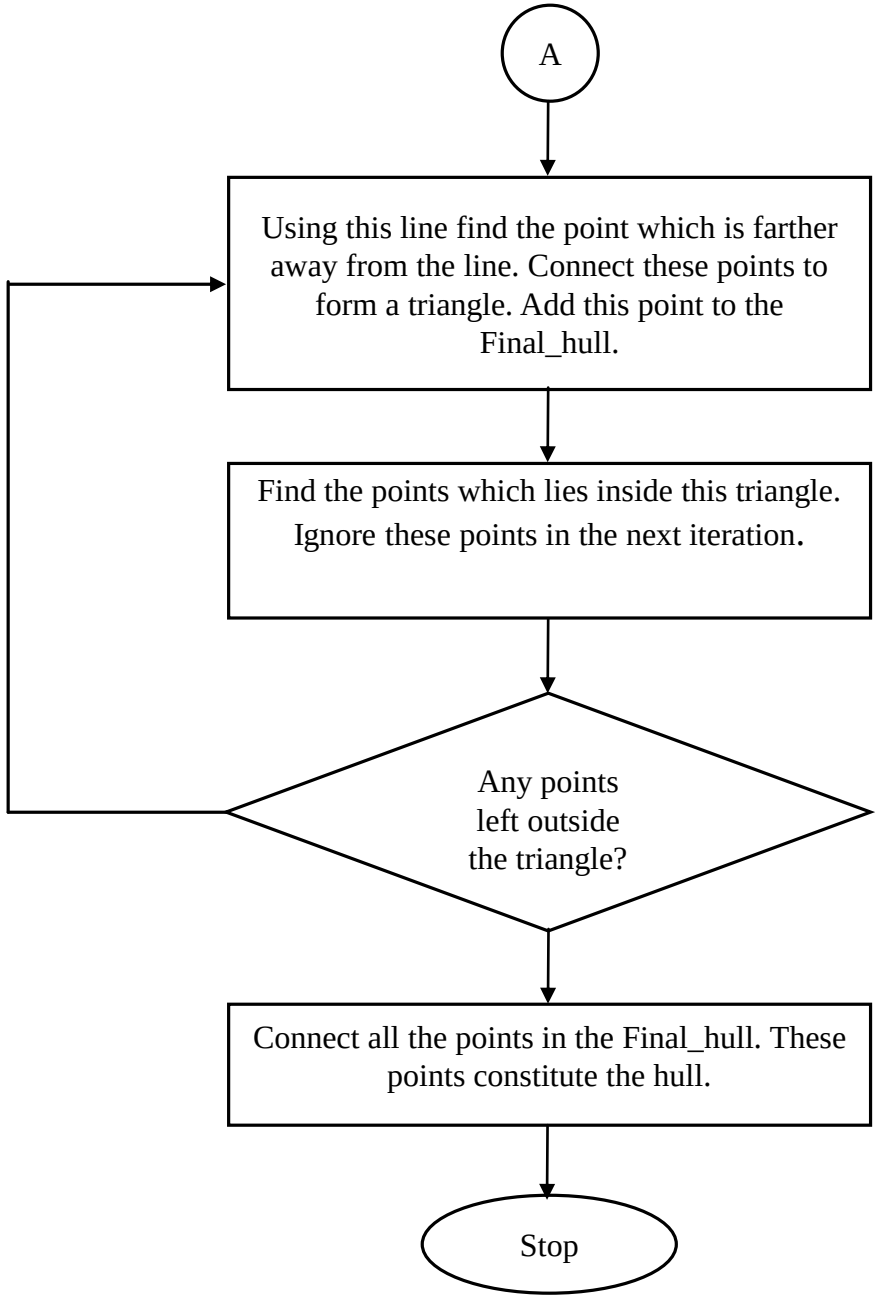


- Step 5: Repeat these steps until no points are left and connect these points.

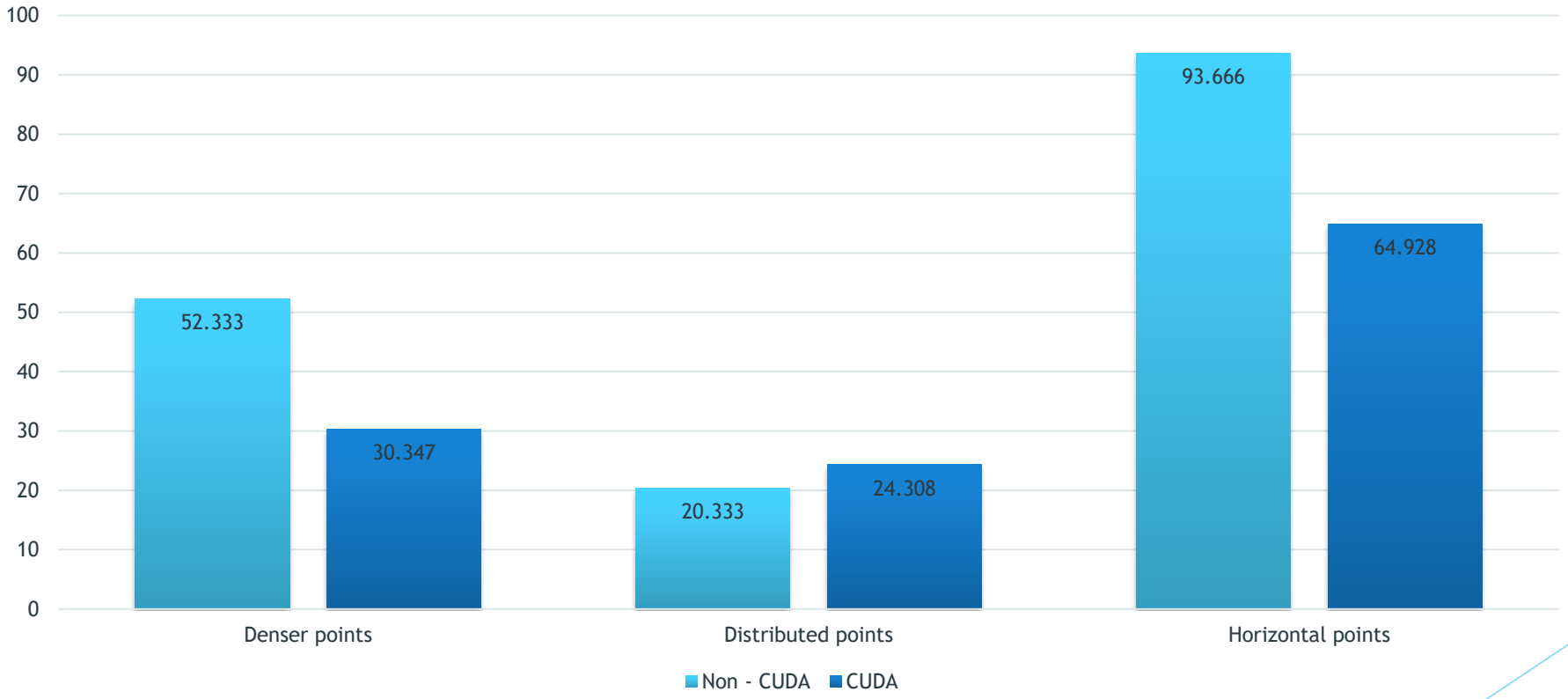


Flowchart for QuickHull Algorithm

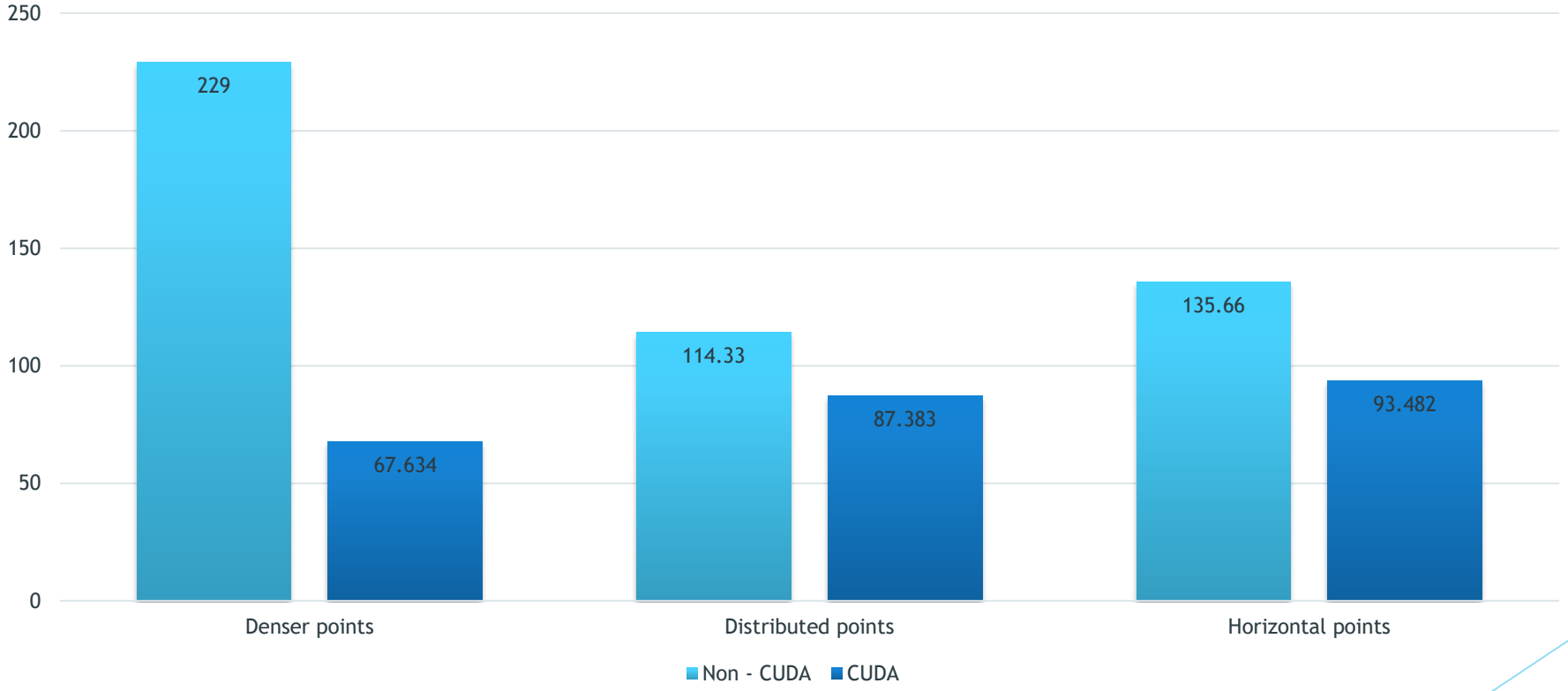



A
Using this line find the point which is farther away from the line. Connect these points to form a triangle. Add this point to the Final_hull.
Find the points which lies inside this triangle. Ignore these points in the next iteration.
Any points left outside the triangle?
Connect all the points in the Final_hull. These points constitute the hull.
Stop

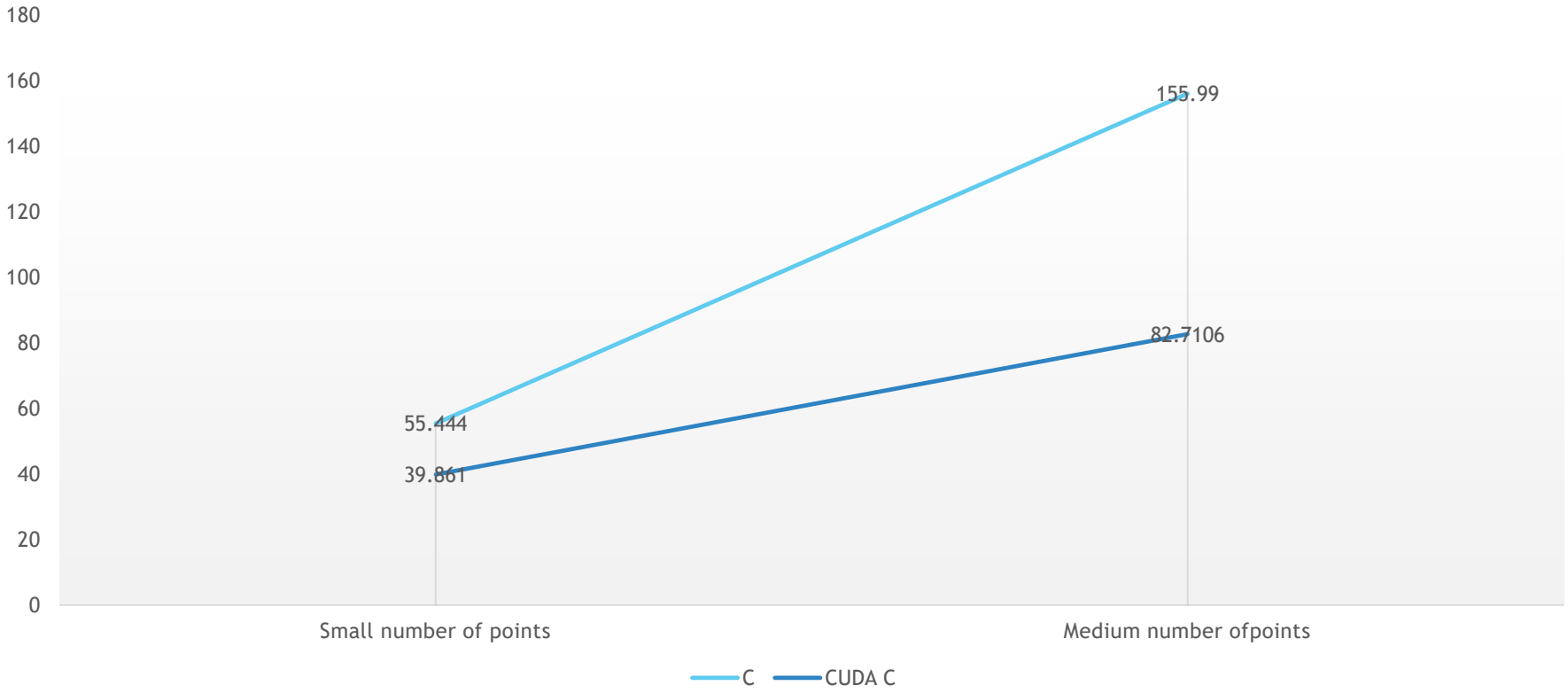
Graph of execution time for Quickhull C and CUDA C program for smaller number of points



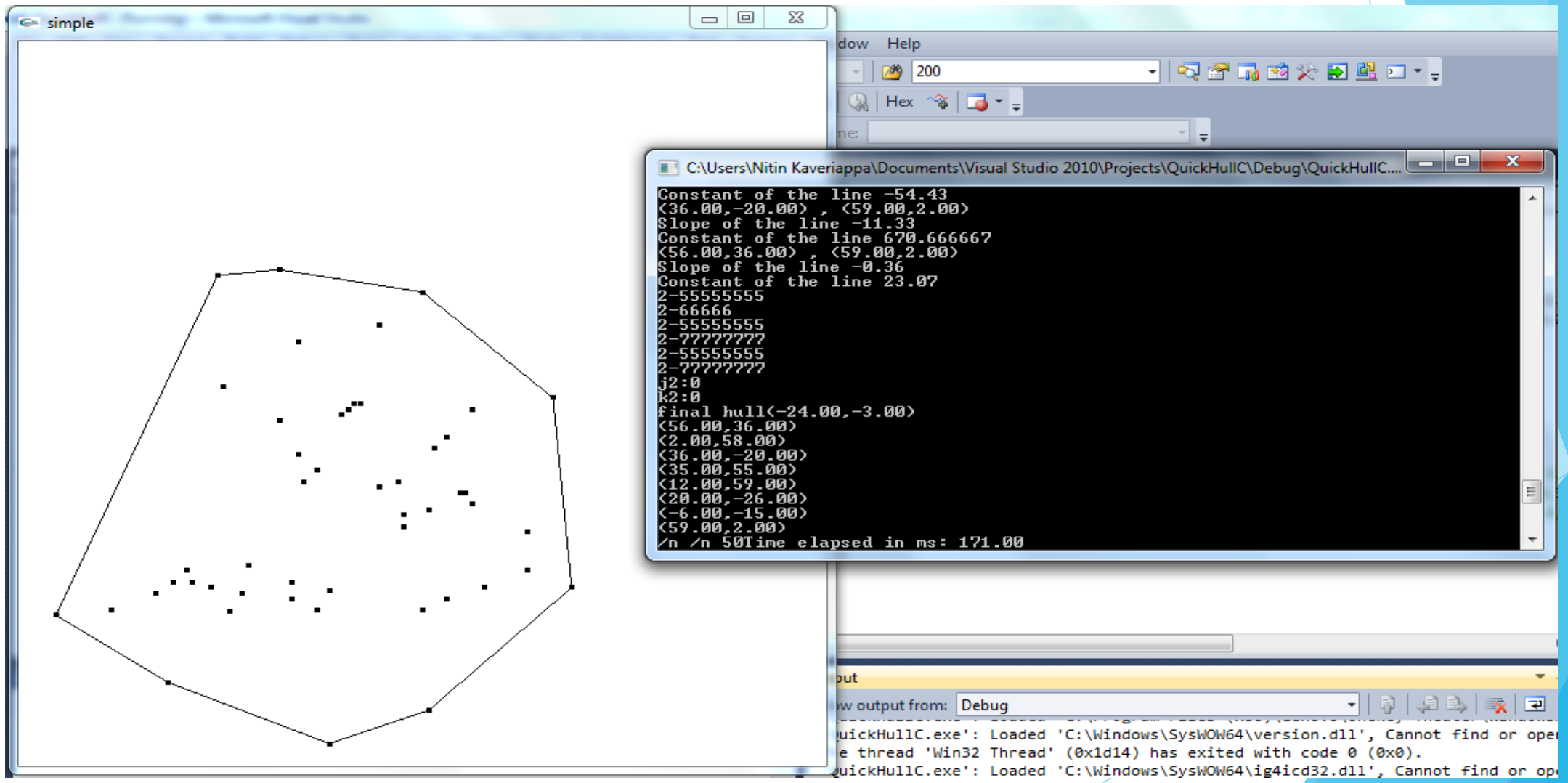
Graph of execution time for Quickhull C and CUDA C program for medium number of points



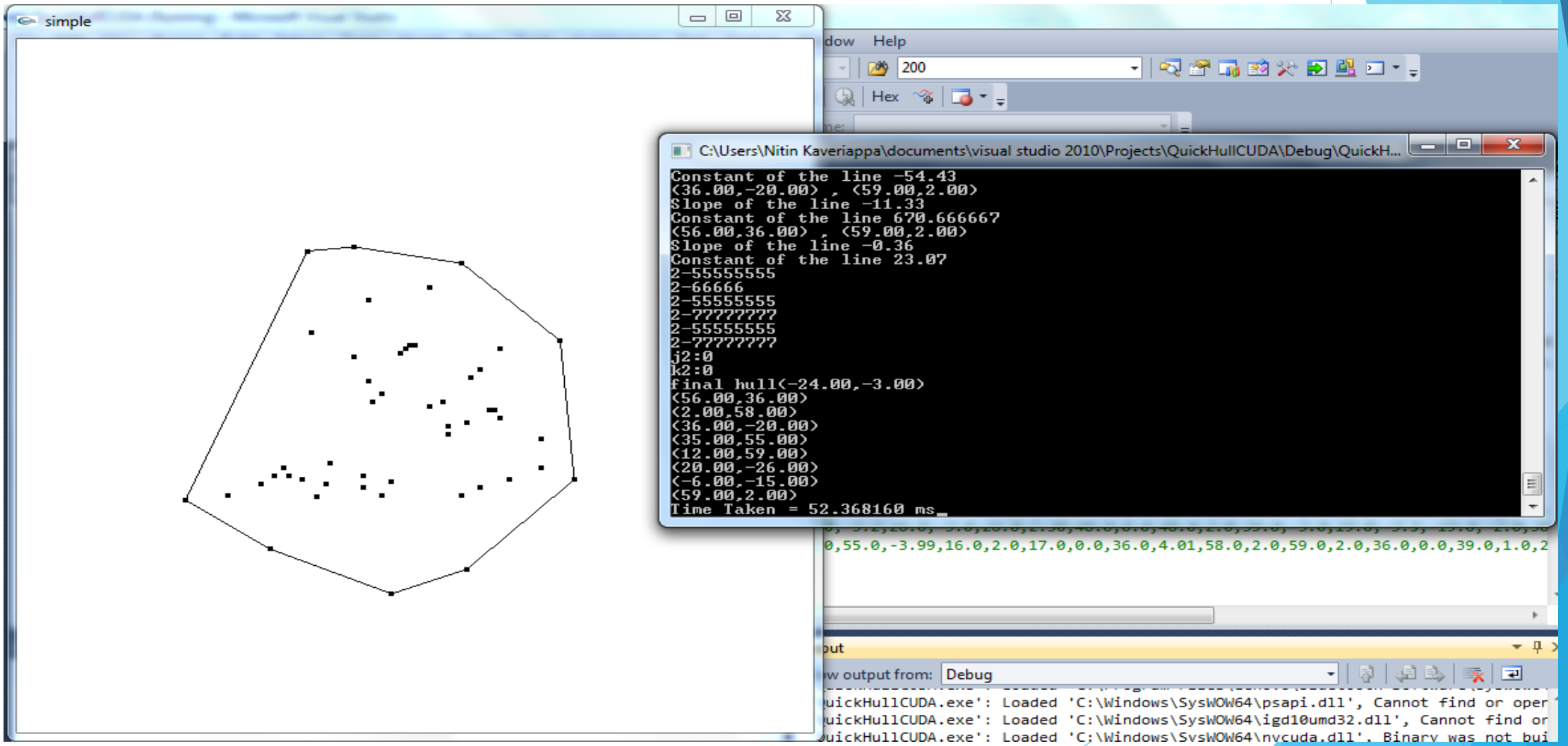
Graph of execution time for Quickhull C and Quickhull CUDA C program



Non-CUDA Output



CUDA Output


simple
dow Help
200
Hex
C:\Users\Nitin Kaveriappa\documents\visual studio 2010\Projects\QuickHullCUDA\Debug\QuickH...
Constant of the line -54.43
(36.00,-20.00) , (59.00,2.00)
Slope of the line -11.33
Constant of the line 670.666667
(56.00,36.00) , (59.00,2.00)
Slope of the line -0.36
Constant of the line 23.07
2-55555555
2-66666
2-55555555
2-77777777
2-55555555
2-77777777
j2:0
k2:0
final hull(-24.00,-3.00)
(56.00,36.00)
(2.00,58.00)
(36.00,-20.00)
(35.00,55.00)
(12.00,59.00)
(20.00,-26.00)
(-6.00,-15.00)
(59.00,2.00)
Time Taken = 52.368160 ms
0,55.0,-3.99,16.0,2.0,17.0,0.0,36.0,4.01,58.0,2.0,59.0,2.0,36.0,0.0,39.0,1.0,2
ut
w output from: Debug
uickHullCUDA.exe': Loaded 'C:\Windows\SysWOW64\psapi.dll', Cannot find or oper
uickHullCUDA.exe': Loaded 'C:\Windows\SysWOW64\igd10umd32.dll', Cannot find or
uickHullCUDA.exe': Loaded 'C:\Windows\SysWOW64\nvcuda.dll', Binary was not bui

Introduction to Smith-Waterman Algortihm

- Optimal local alignment of two sequences
- Performs an exhaustive search for the optimal local Alignment
- Based on the 'dynamic programming' algorithm
  - ✓ Fill the matrix
  - ✓ Find the maximal value (score) in the matrix
  - ✓ Trace back from the score until a 0 value is reached

Smithwaterman Algorithm

- Initialisation: a,b->Strings over the alphabet ∑
  m->length of (a)
  n->length of (b)
  H(i,j)->is the max similarity score between a suffix of a[1....i] and a suffix of b[1....j]
  W(c,d), c, d ϵ ∑ U{‘-‘},’-‘ is the gap scoring scheme
- Steps: Matrix H is built as follows:
  H(i,0)=0, 0 <= i <=m
  H(0,j)=0, 0 <=j<=n
  If ai=bj, then W(ai,bj)=W(match) or
  If ai!=bj, then W(ai,bj)=W(mismatch)

H(i,j)= max { H(i-1,j-1+w(ai,bj) Match/mismatch
H(i-1,j)+W(ai,-) Deletion
H(i,j-1)+W(-,bj) Insertion } for
1<=i<=m and 1<=j<=n
Where W (match)=+2
W(a,-)=W(-,b)=W(mismatch)=-1

- Output: Scoring matrix H

	-	A	C	A	C	C	A	C	A
-	0	0	0	0	0	0	0	0	0
A	0	2	1	2					
G	0	1	1	1					
C	0	0	3	2					
A	0								
C	0								
A	0								
C	0								
A	0								

- Input: Scoring matrix H
- Output: Alignment matrix T

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{pmatrix}$$

- Input: Alignment matrix T.
- Output: Optimal local alignment of sequences a and b i.e. for determining similar regions between two strings.

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{pmatrix}$$

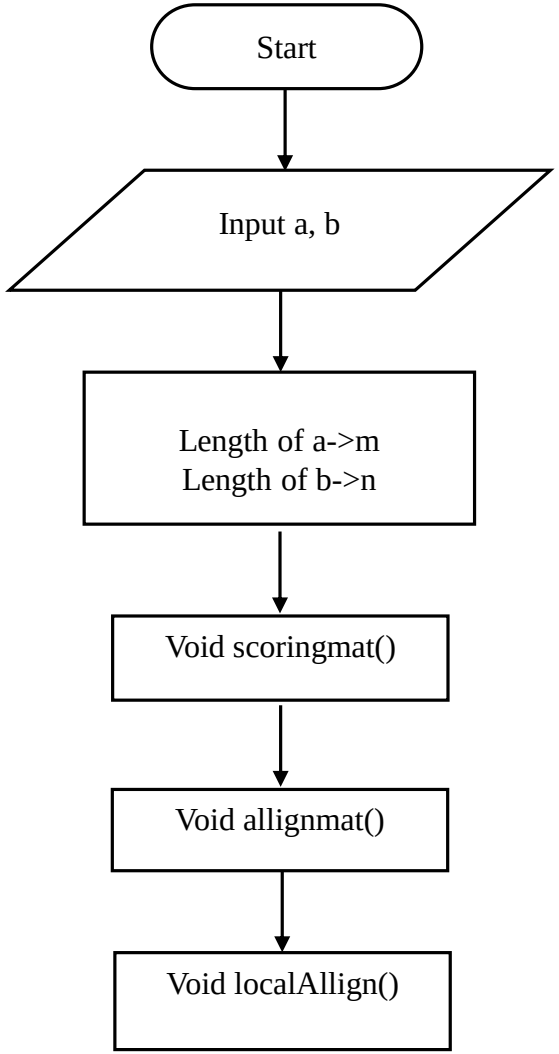
$$T = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{pmatrix}$$

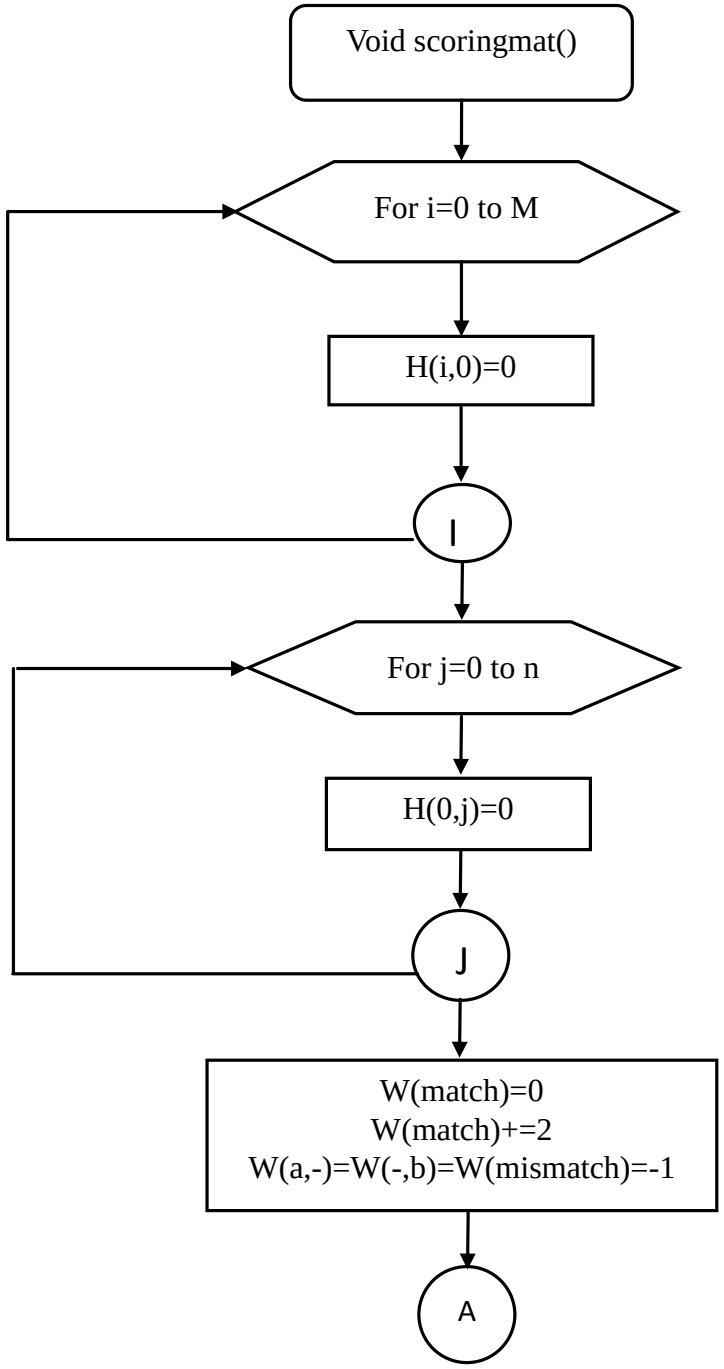
For the example, the results are:

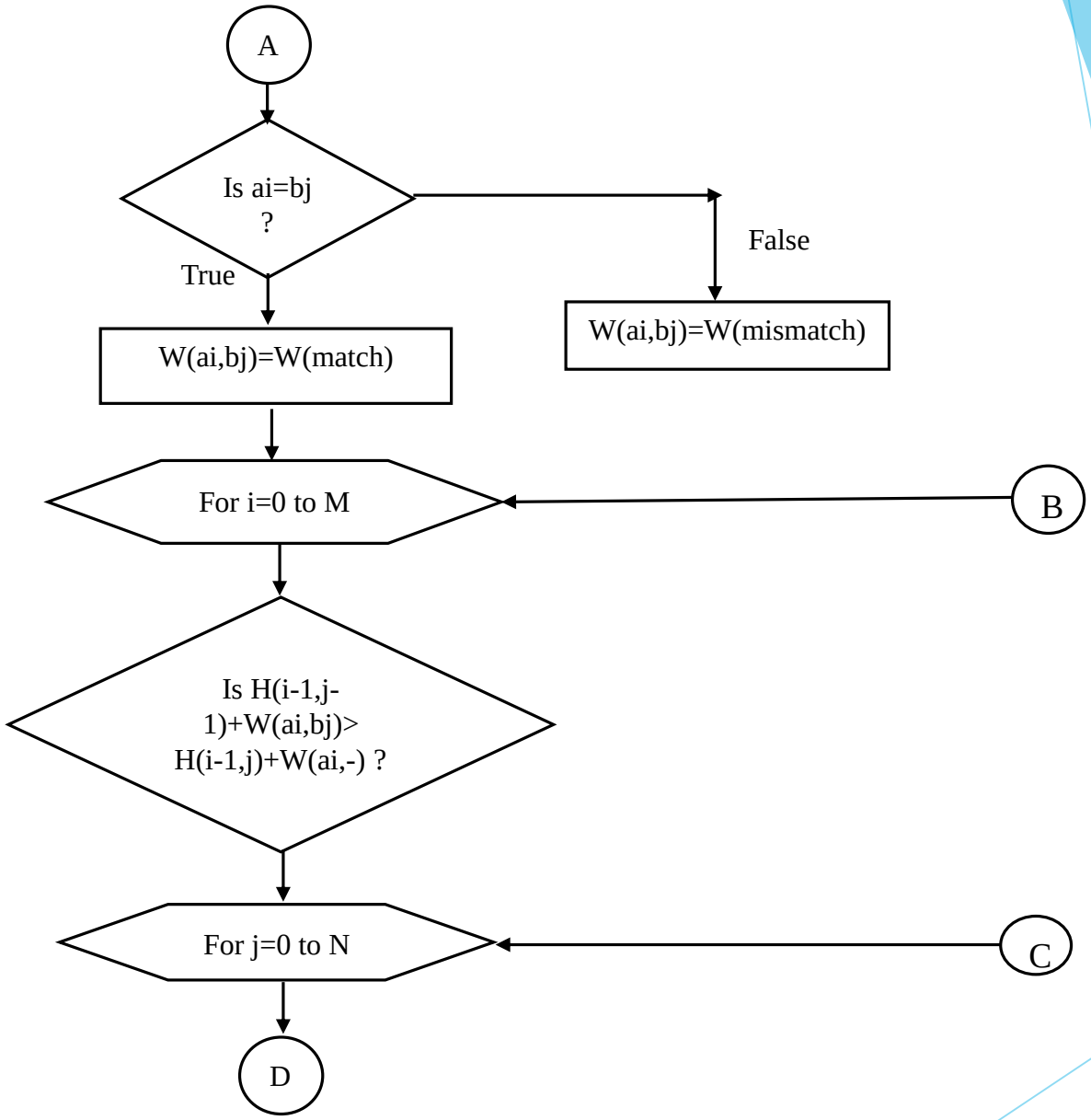
Sequence 1 = A-CACACTA

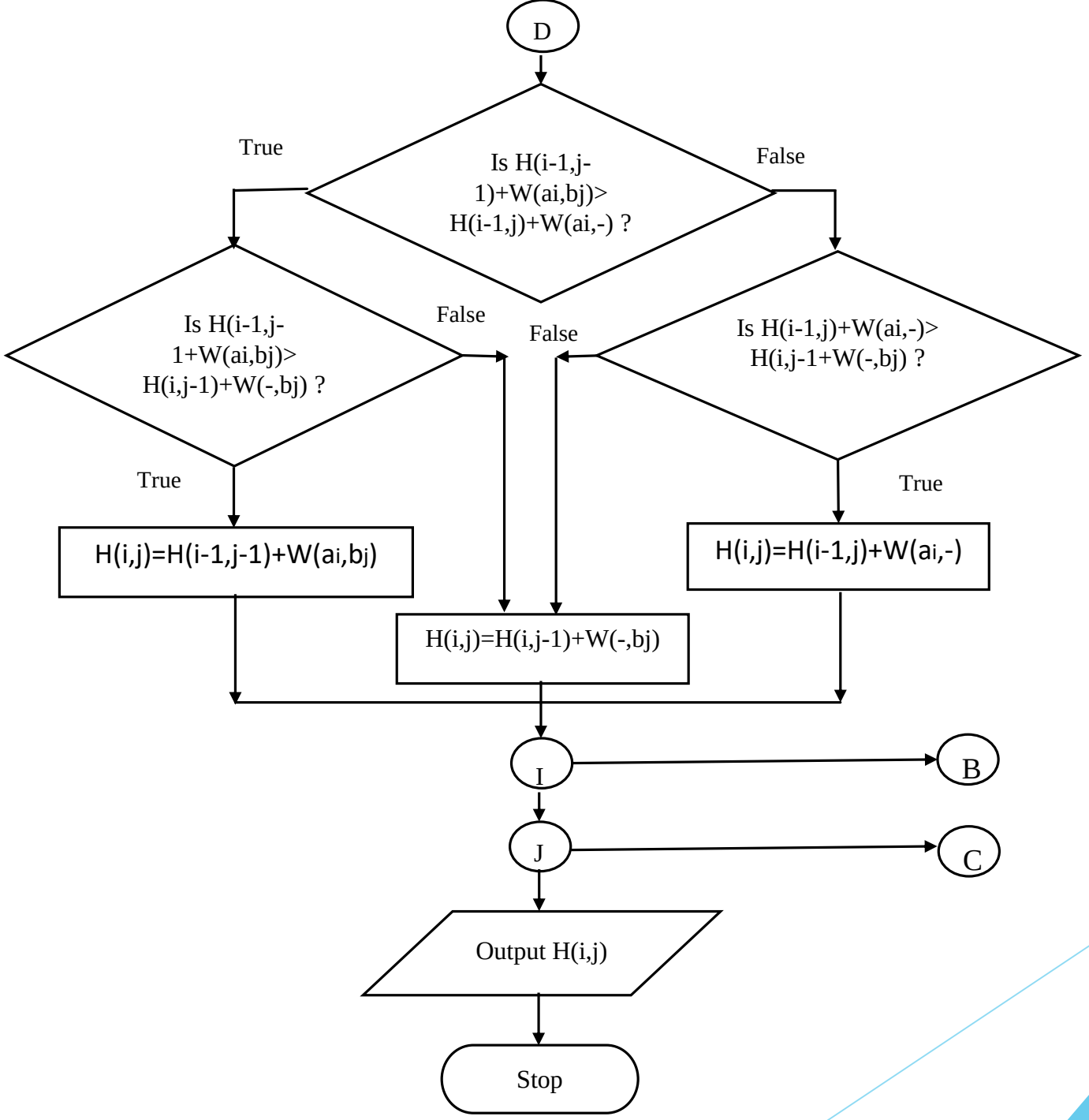
Sequence 2 = AGCACAC-A

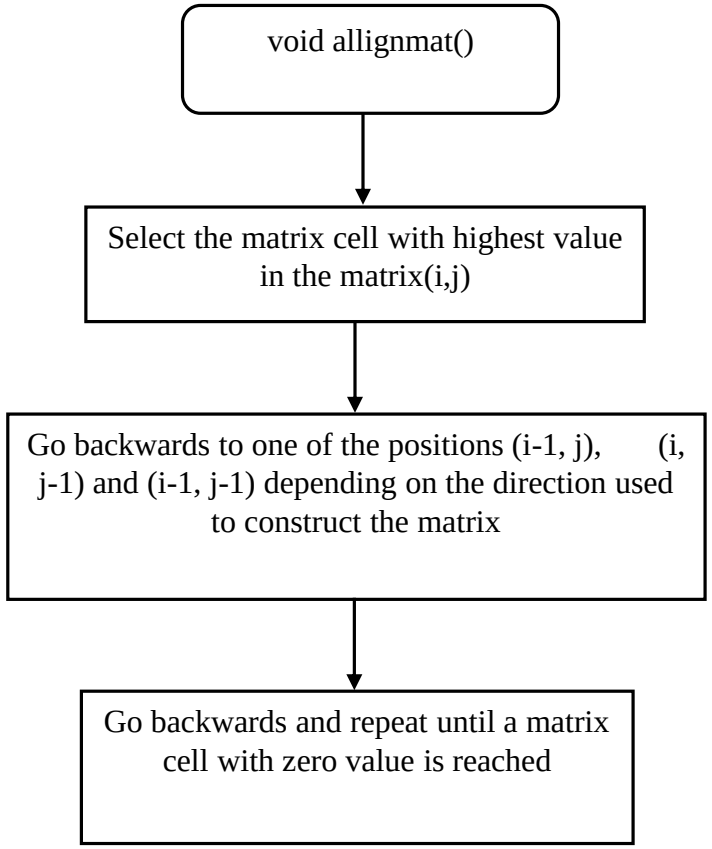
Flowchart for Smith Waterman Algorithm

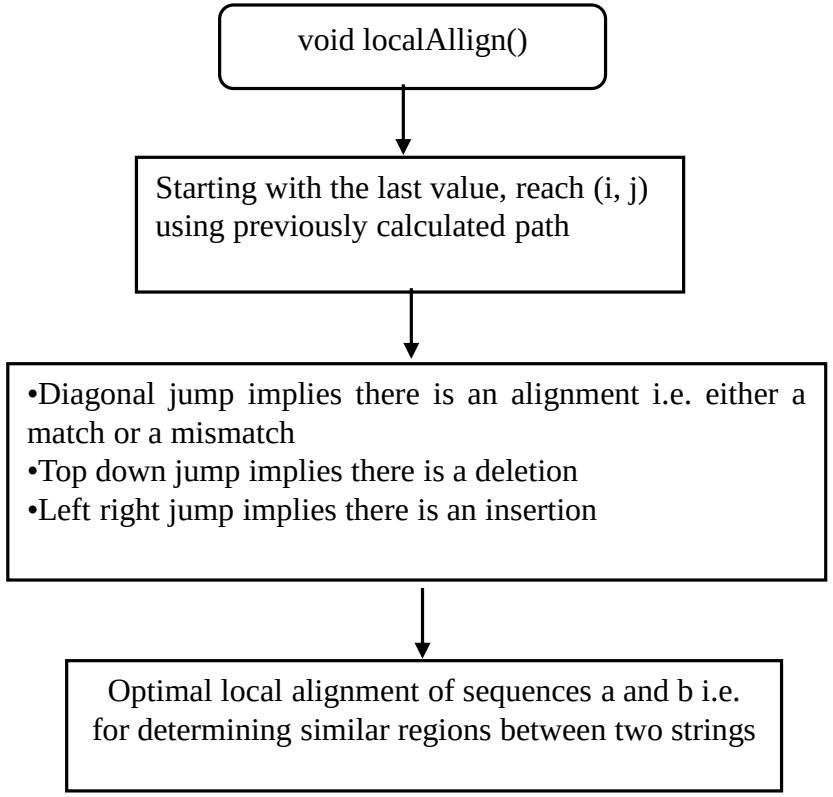



Void scoringmat()
For i=0 to M
H(i,0)=0
I
For j=0 to n
H(0,j)=0
J
W(match)=0
W(match)+=2
W(a,-)=W(-,b)=W(mismatch)=-1
A

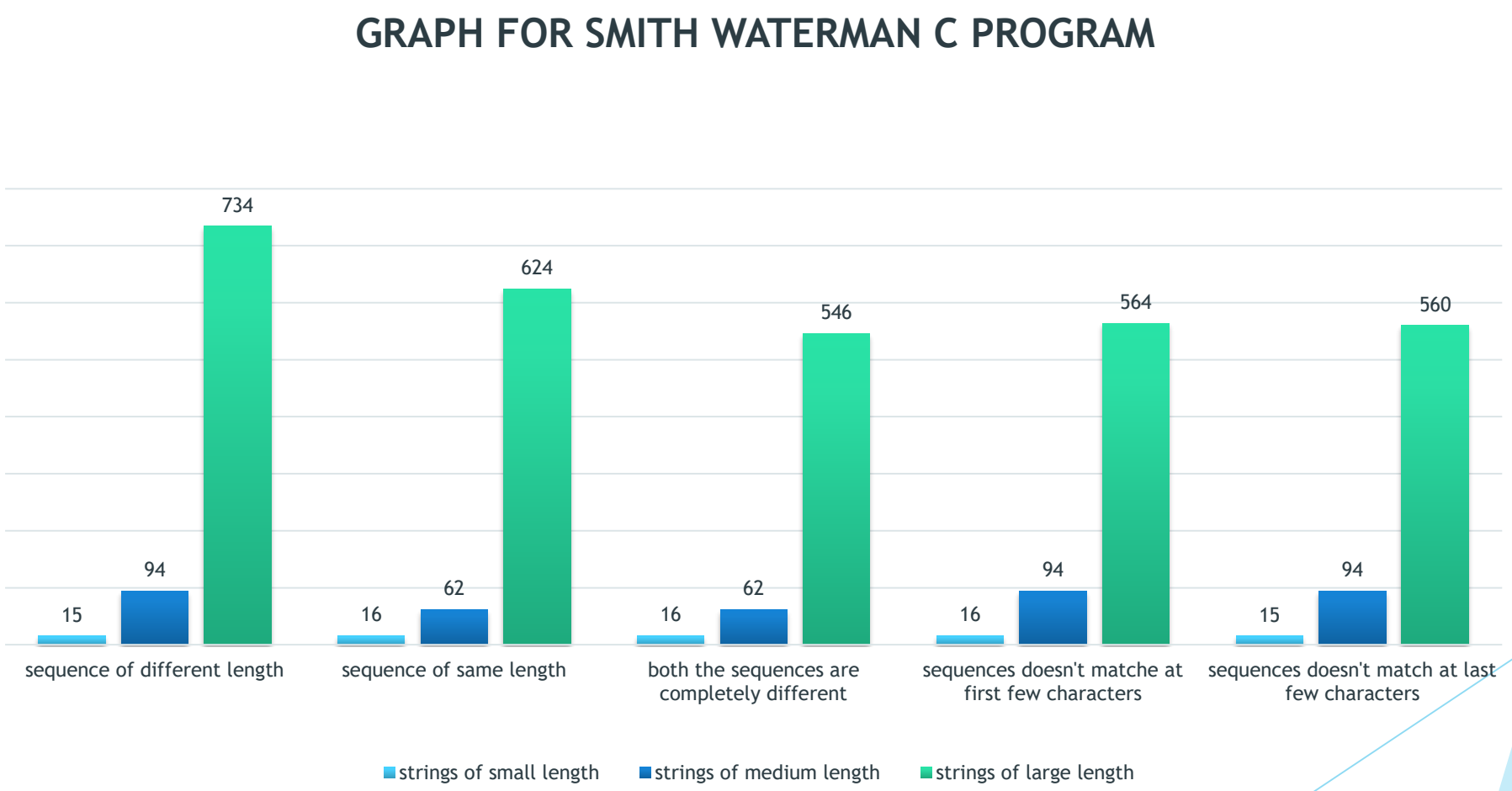

A
Is ai=bj
?
True
False
W(ai,bj)=W(match)
W(ai,bj)=W(mismatch)
For i=0 to M
B
Is H(i-1,j-
1)+W(ai,bj)>
H(i-1,j)+W(ai,-) ?
For j=0 to N
C
D


D
Is H(i-1,j-1)+W(ai,bj)> H(i-1,j)+W(ai,-) ?
True
False
Is H(i-1,j-1+W(ai,bj)> H(i,j-1)+W(-,bj) ?
False
Is H(i-1,j)+W(ai,-)> H(i,j-1+W(-,bj) ?
False
True
True
H(i,j)=H(i-1,j-1)+W(ai,bj)
H(i,j)=H(i-1,j)+W(ai,-)
H(i,j)=H(i,j-1)+W(-,bj)
I
B
J
C
Output H(i,j)
Stop

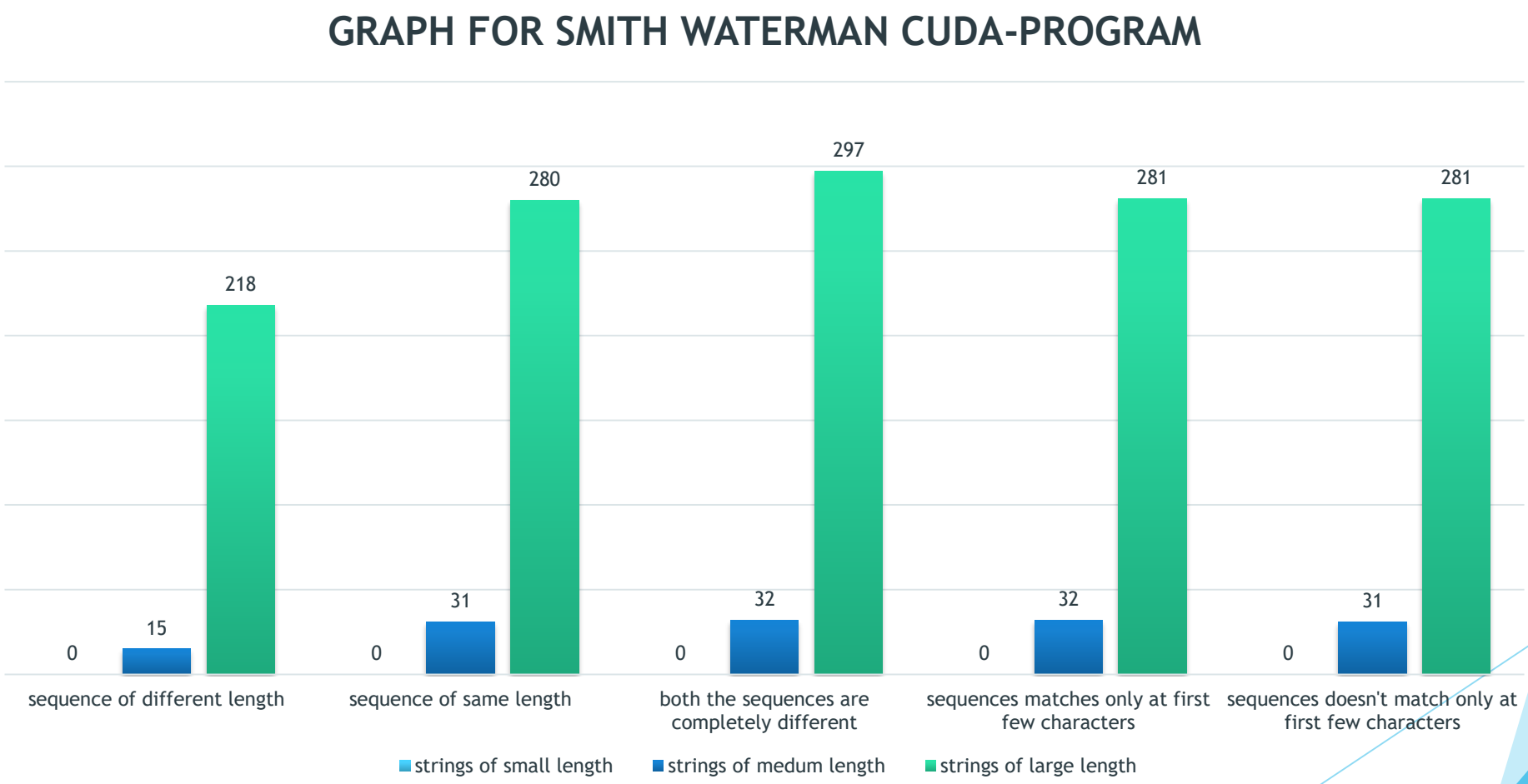

void allignmat()
Select the matrix cell with highest value in the matrix(i,j)
Go backwards to one of the positions (i-1, j), (i, j-1) and (i-1, j-1) depending on the direction used to construct the matrix
Go backwards and repeat until a matrix cell with zero value is reached


void localAllign()
Starting with the last value, reach (i, j) using previously calculated path
•Diagonal jump implies there is an alignment i.e. either a match or a mismatch
•Top down jump implies there is a deletion
•Left right jump implies there is an insertion
Optimal local alignment of sequences a and b i.e. for determining similar regions between two strings

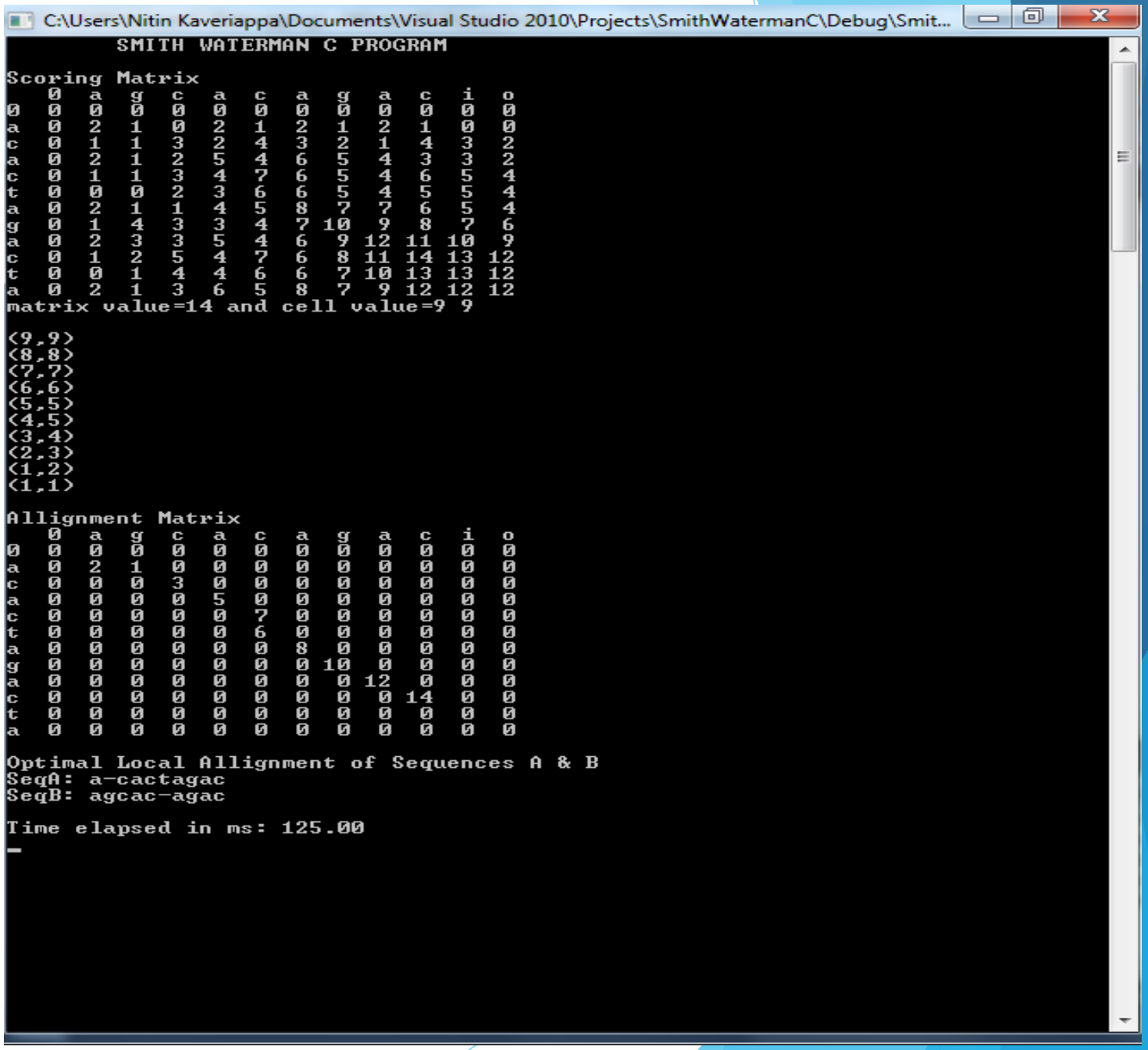
Non-CUDA Performance Graph



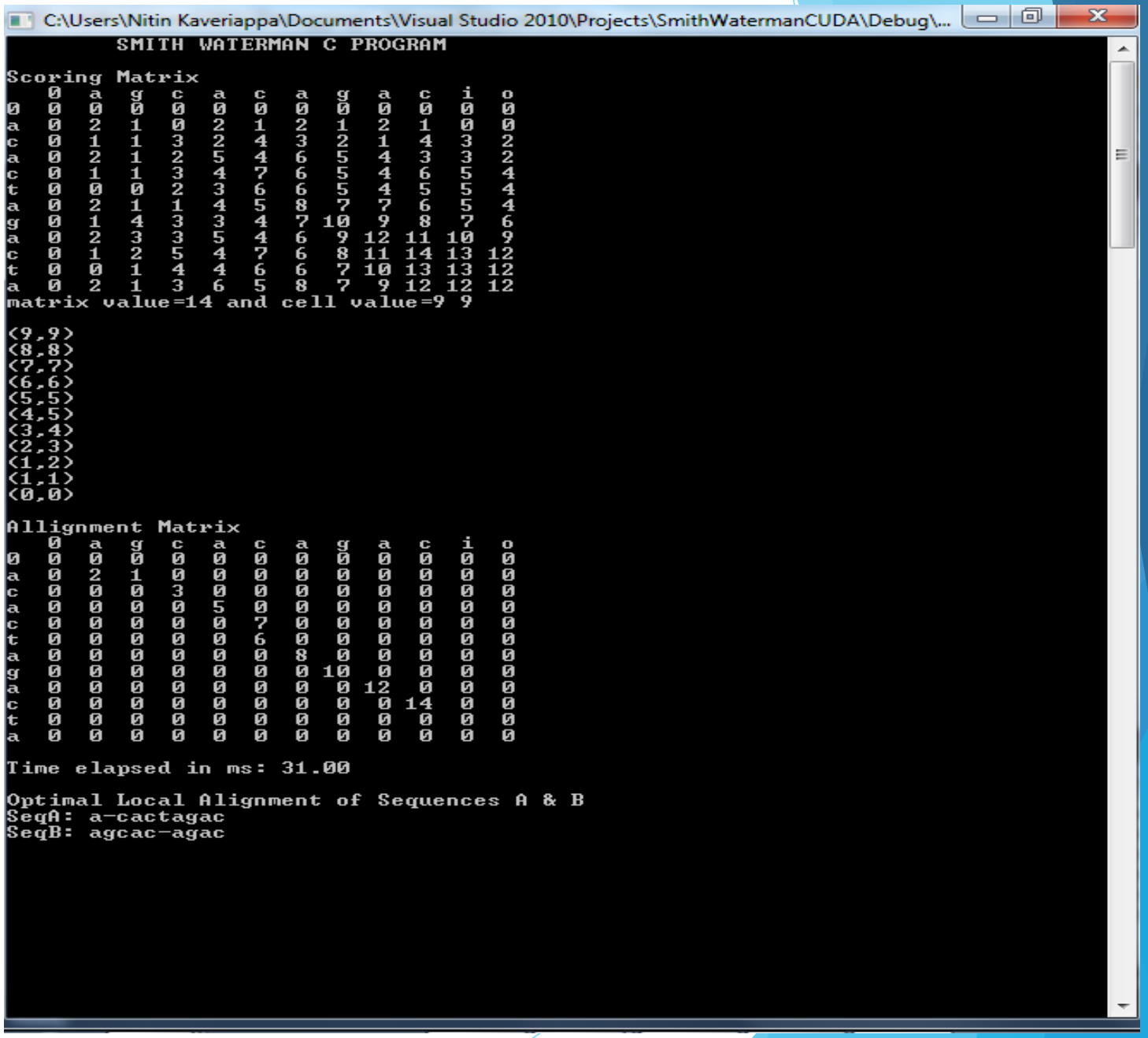
CUDA Performance Graph



Non-CUDA Output



CUDA Output


C:\Users\Nitin Kaveriappa\Documents\Visual Studio 2010\Projects\SmithWatermanCUDA\Debug\...
SMITH WATERMAN C PROGRAM
Scoring Matrix
matrix value=14 and cell value=9 9
(9,9)
(8,8)
(7,7)
(6,6)
(5,5)
(4,5)
(3,4)
(2,3)
(1,2)
(1,1)
(0,0)
Allignment Matrix
Time elapsed in ms: 31.00
Optimal Local Alignment of Sequences A & B
SeqA: a-cactagac
SeqB: agcac-agac

Conclusion

- The use of CUDA for small inputs may actually increase the execution time rather than decreasing it because of the time taken for memory transfers between the CPU and GPU
- As in the CPU the GPU also has a hardware limit known as the computational capability which is determined by the number of cores contained in the GPU. This restricts the maximum size of the input or number of iterations
- We have observed that as the computational need of a program increases the execution time decreases when compared with CPU vs GPU, hence improving the overall performance program execution

Thank You.