\*\*University of Pennsylvania, CIS 565: GPU Programming and Architecture,

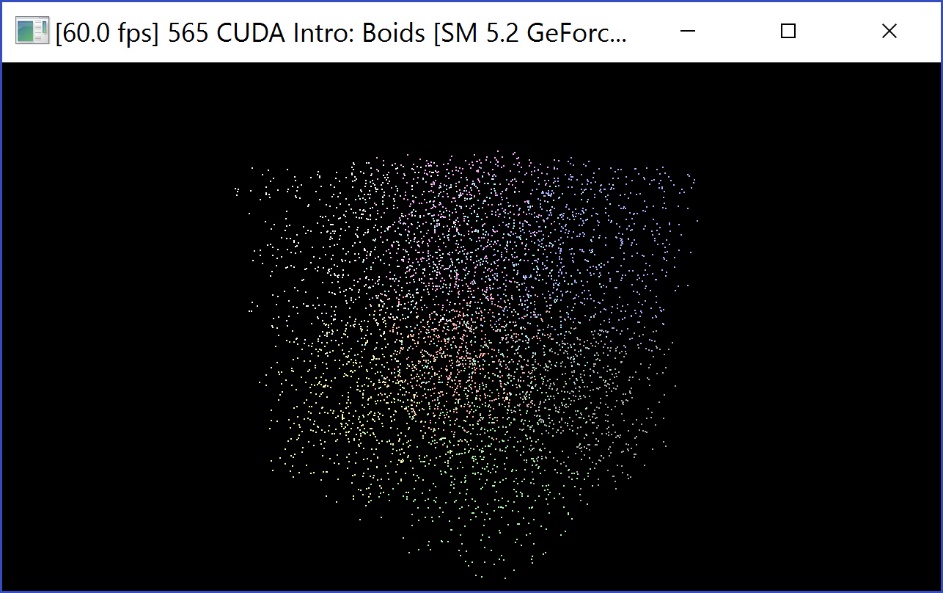
Project 1 - Flocking\*\*

\* YAOYI BAI

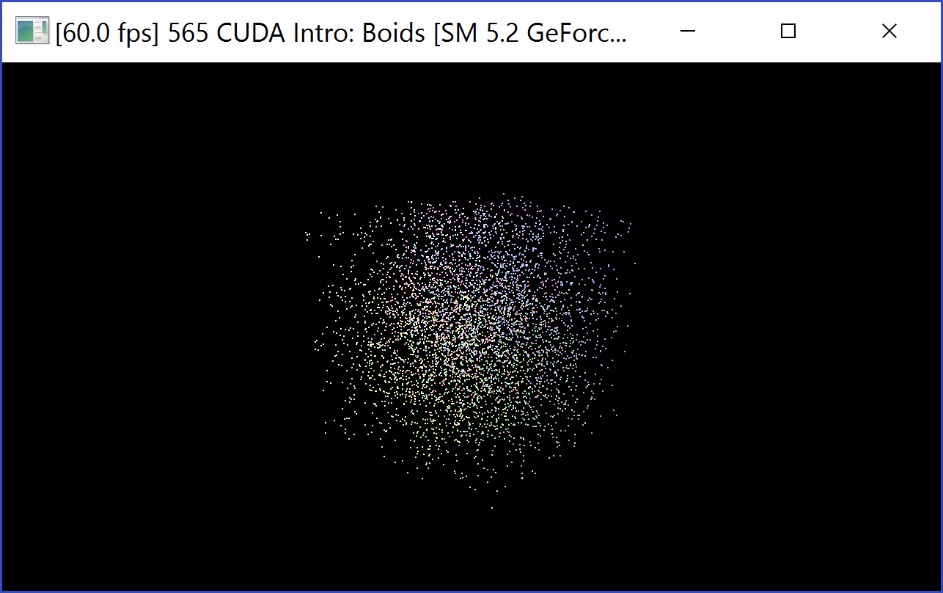
\* Tested on: Windows 10 Pro, i7-6700HQ @ 2.60GHz 16GB, GTX 980M 4GB (personal)

### YAOYI BAI

Screenshot:



image\_1



image\_2

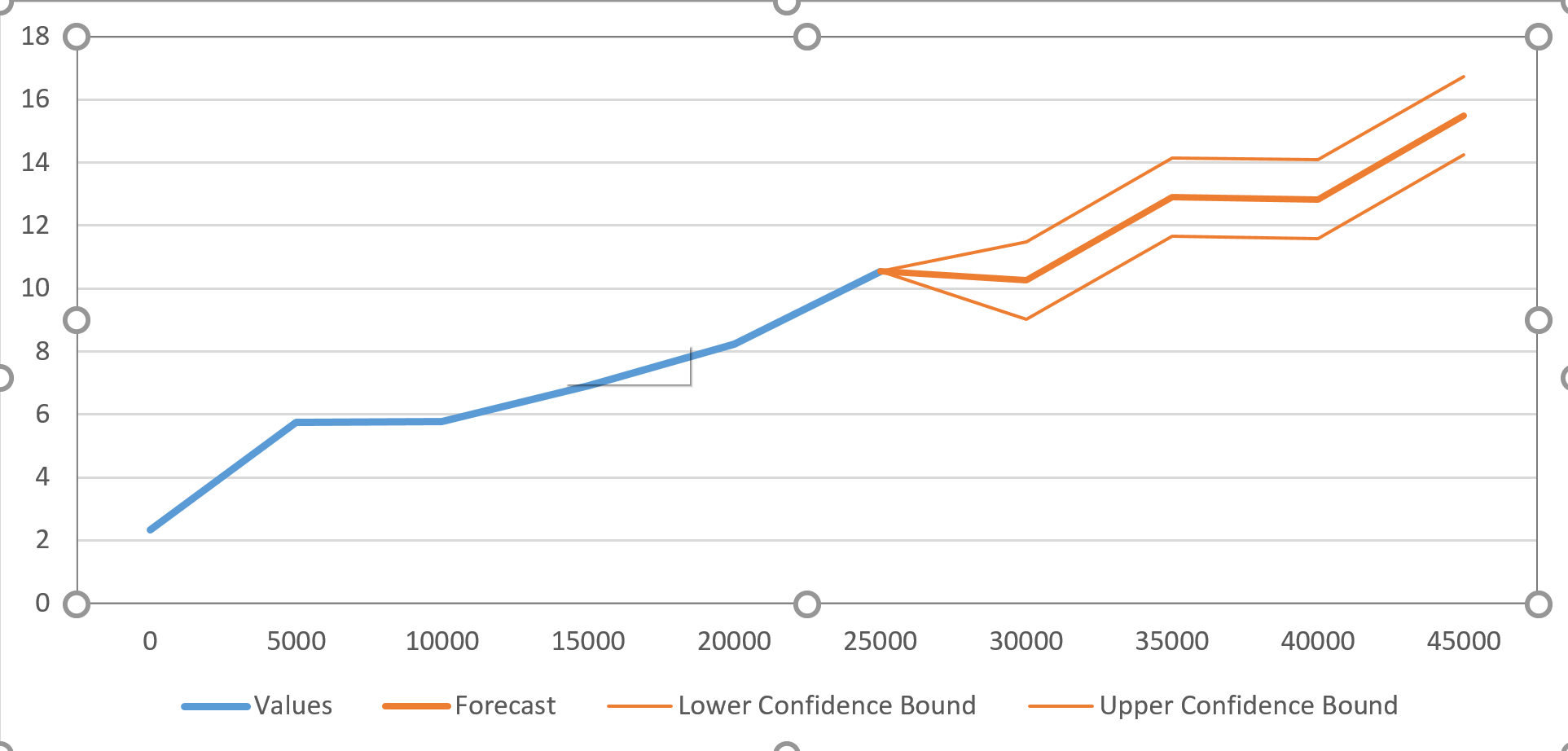


image\_3

**Analysis**

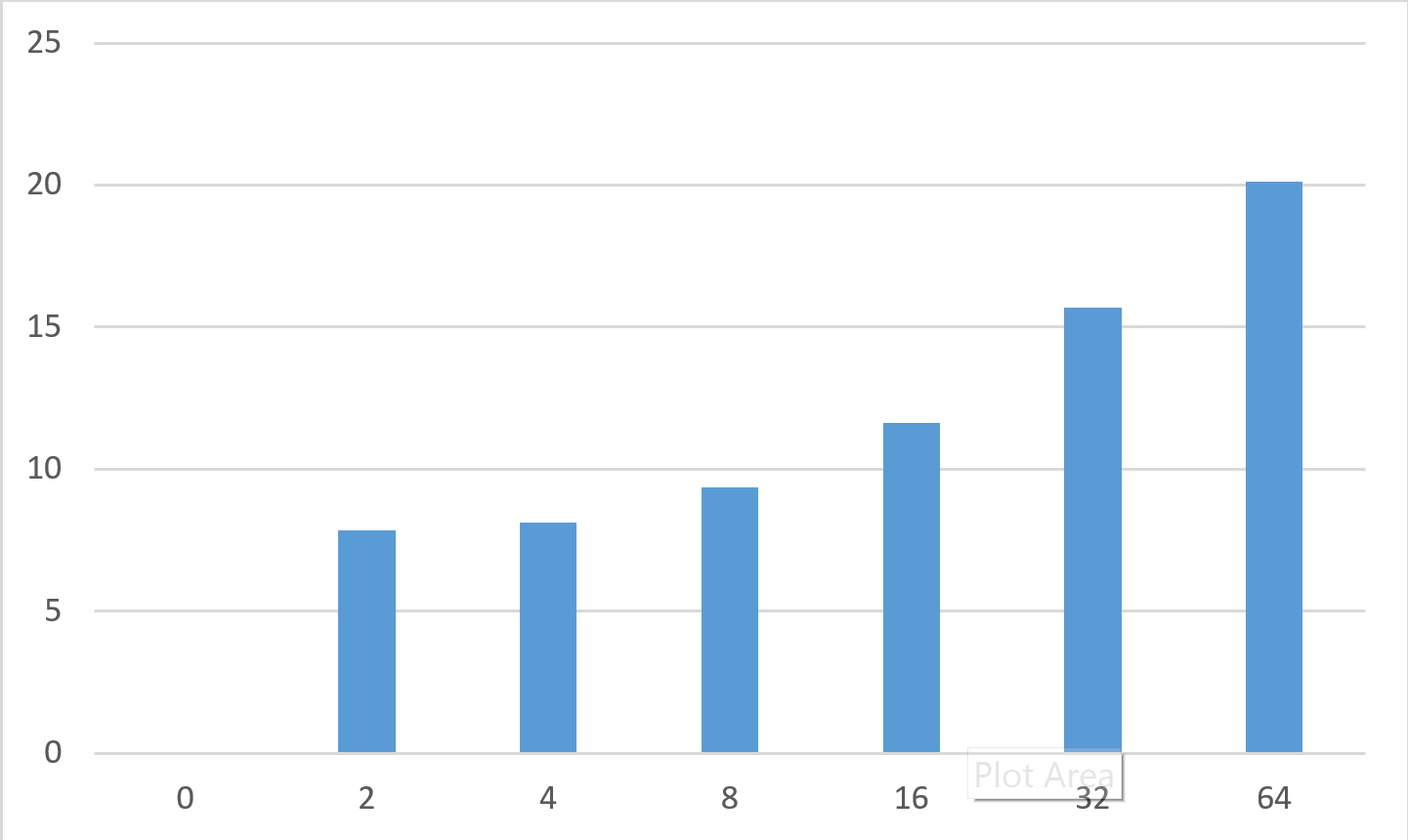
Graph1:

The x-axis of this graph means the number of particles, whereas the y-axis means the time we spend on simulation. And we can predict that when the number of particles increases, the time we will spend on the simulation. We can see that the graph is basically linear when the number of particle is not huge enough. The time for simulation is not only related to the number of particles, but also has something to do with the capacity of the computer that we are working on. There will be a threshold in the future so that the simulation time will increase greatly when the number of particles increases, because of the calculation ability of the computer and to be more specific the device.



Graph2:

The x-axis of this graph means the size of cells, whereas the y-axis means the time we spend on simulation. We can see that the time we will spend on simulation will dramatically increase when the size of the cell increases by multiplying 2 every time, which means we will have to search for more boids inside one cell, and of course we will have to search for more boids inside other three cells. Therefore, it will take more time for the simulation.



**Other issue about the algorithm**

Algorithm: How to detect the neighborhood 2D and 3D cell

**Step1: define the number of any neighborhood cells as:**

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

**Step2: calculate which cells are neighborhood cells to these 16 cells:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| places | A(0) | B(1) | C(2) | D(3) | E(4) | F(5) | G(6) | H(7) |
| 0 | NULL | NULL | NULL | NULL | 1 | NULL | 4 | 5 |
| 1 | NULL | NULL | NULL | 0 | 2 | 4 | 5 | 6 |
| 2 | NULL | NULL | NULL | 1 | 3 | 5 | 6 | 7 |
| 3 | NULL | NULL | NULL | 2 | NULL | 6 | 7 | NULL |
| 4 | NULL | 0 | 1 | NULL | 5 | NULL | 8 | 9 |
| 5 | 0 | 1 | 2 | 4 | 6 | 8 | 9 | 10 |
| 6 | 1 | 2 | 3 | 5 | 7 | 9 | 10 | 11 |
| 7 | 2 | 3 | NULL | 6 | NULL | 10 | 11 | NULL |
| 8 | NULL | 4 | 5 | NULL | 9 | NULL | 12 | 13 |
| 9 | 4 | 5 | 6 | 8 | 10 | 12 | 13 | 14 |
| 10 | 5 | 6 | 7 | 9 | 11 | 13 | 14 | 15 |
| 11 | 6 | 7 | NULL | 10 | NULL | 14 | 15 | NULL |
| 12 | NULL | 8 | 9 | NULL | 13 | NULL | NULL | NULL |
| 13 | 8 | 9 | 10 | 12 | 14 | NULL | NULL | NULL |
| 14 | 9 | 10 | 11 | 13 | 15 | NULL | NULL | NULL |
| 15 | 10 | 11 | NULL | 14 | NULL | NULL | NULL | NULL |

then we have the table (int Neigh\_Ref[16][8]) below:

(the reason we apply a number behind the Arabic words is make where and how we store this table clearer)

**Step3: a boid can lies in 4 kinds of places inside any cell:**

For any position of any boid anytime, according to the definition, we have a variety named dev\_pos.

Then we can calculate the exact position of a boid inside a cell.

//x\_grid stands for the x-position of the 4-by-4 cell matrix

// y\_grid stands for the y-position of the 4-by-4 cell matrix

// x\_grid\_inside stands for the x-position inside a cell (which decides position X, Y, Z or W)

// y\_grid\_inside stands for the y-position inside a cell (which decides position X, Y, Z or W)

Let

int x\_grid = &dev\_pos[0]/girdCellWidth;

int y\_grid = &dev\_pos[1]/girdCellWidth;

int x\_grid\_inside = &dev\_pos[0]\girdCellWidth;

int y\_grid\_inside = &dev\_pos[1]\girdCellWidth;

Then we can define a 16-by-2 array named Cell\_Pos(int Cell\_Pos[16][3]) which stores the position of cells inside a 4-by-4 cell matrix:

|  |  |  |
| --- | --- | --- |
| cell\_x\_pos | cell\_y\_pos | cell\_number |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 2 | 2 |
| 0 | 3 | 3 |
| 1 | 0 | 4 |
| 1 | 1 | 5 |
| 1 | 2 | 6 |
| 1 | 3 | 7 |
| 2 | 0 | 8 |
| 2 | 1 | 9 |
| 2 | 2 | 10 |
| 2 | 3 | 11 |
| 3 | 0 | 12 |
| 3 | 1 | 13 |
| 3 | 2 | 14 |
| 3 | 3 | 15 |

then we compare the result:

if (Cell\_Pos[x\_res][y\_res]={x\_grid,y\_grid}) = TRUE then we know which cell the boid belongs to.

**To avoid contradiction, we assume that any boid only lies on the left and upper side of the cell if this boid happens to lie on any intersection of two cells or on the vertexes of cells.**

Then we can calculate the which cell the boid lies inside if we check the third column of Cell\_Pos array or simply calculate:

int Cell\_Num = y\_pos+(x\_pos)\*4;

After that, we have to calculate which part inside a cell the boid lies in;

Assume the boid can be divided into four parts X, Y, Z and W

|  |  |
| --- | --- |
| X | Y |
| Z | W |

There are 9 kinds of situations where the boid lies in, as a result, there are also 9 kinds of cases that the neighborhood cells should be:

1. If the boid lies in X in a cell: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, A, B, D;

When

0<=x\_grid\_inside<0.5 && 0<=y\_grid\_inside<0.5

1. If the boid lies in Yin a cell: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, B, C, E;

When

0.5<x\_grid\_inside<1 && 0<=y\_grid\_inside<0.5

1. If the boid lies in Z in a cell: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, D, F, G;

When

0<=x\_grid\_inside<0.5 && 0.5<y\_grid\_inside<1

1. If the boid lies in W in a cell: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, E, G, H;

When

0.5<x\_grid\_inside<1 && 0.5<y\_grid\_inside<1

1. If the boid lies on the intersection between X and Y: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, B;

When

0<=x\_grid\_inside<0.5 && y\_grid\_inside=0.5

1. If the boid lies on the intersection between X and Z: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, D;

When

x\_grid\_inside=0.5 && 0<=y\_grid\_inside<0.5

1. If the boid lies on the intersection between Y and W: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, E;

When

x\_grid\_inside=0.5 && 0.5<y\_grid\_inside<1

1. If the boid lies on the intersection between Z and W: then the four cells we should check in the uniform spatial grid algorithm would be: this cell, G;

When

0.5<x\_grid\_inside<1 && y\_grid\_inside=0.5

1. If the boid lies in the center of the cell: then the four cells we should check in the uniform spatial grid algorithm would be: only this cell.

When

x\_grid\_inside=0.5 && y\_grid\_inside=0.5

**Step4: Then we check the four neighborhood cells of every boid by referring to the table(array) Neigh\_Ref, and we know the exact number of the cells we have to check. So that we can proceed the algorithm.**

To use parallel calculation and speed up the algorithm, we have to put all of these varieties into device (GPU), and make the calculation above into kernel functions.

**Step5: We should extend the algorithm into 3D.**

First we should divide a cell into 8 parts and calculate which part the boid lies inside according to a similar calculating in Step3.

There are 8 kinds of neighborhood cells depending on which part of cell the boid lies inside. We can calculate the number of these cells by add one or minus one from the x, y and z coordinates of the cell we are working on.

Thereby, the algorithm is more correct.