Adding Heterogenous Computing Support to MASS

Ali Arslan Yousaf

Summary Report of work done as part of
CSS 700: Master’s Thesis

Master of Science in Computer Science & Software Engineering
University of Washington, Bothell
Autumn, 2019

Committee:
Dr. Munehiro Fukuda, Chair
Dr. Michael Stiber, Committee Member
Dr. Erika Parsons, Committee Member
Introduction
MASS, or Multi-Agent Spatial Simulation, is a framework for high performance computing and big data processing. It applies code migration to present an agent-based modelling paradigm where mobile agents move over a set of spaces to achieve computational goals. Initially available as a C++ and Java library, MASS has recently been ported to CUDA to take advantage of the massive performance gains available via GPU computing.

However, the memory on most commodity GPUs is fairly limited which serves as a distraction and imposes strict limits on the problem sizes which can be computed with MASS CUDA. This project aims to enable heterogeneous CPU-GPU computing support in MASS which would allow the framework to work on far larger problem sets (while graphics memory is generally limited to around 12 GB, DRAM can scale to terabytes). Due to the nature of the problems most well suited to MASS, many of which tend to involve spatial spreads of inter-dependent data, this poses significant challenges. For example, if a certain part of the problem space is loaded into the GPU memory, and border information needs to be updated, it would require the neighbouring set of data to be loaded which would result in latency issues and programmer productivity decline due to the need for extensive manual memory management.

Hence, this project aims to implement a scheme which exploits CPU-GPU heterogeneous computing specifically for Agent Based Multi-Processing to allow the majority of problems suited to MASS to be efficiently run on GPUs with oversubscribed memory with no changes on the part of the programmer.

Pivot and updated direction from thesis plan
This section describes the direction of the project and the changes to the overall specifications from the thesis plan filed at the beginning of the quarter.

The tentative goals of the project as laid out in the thesis proposal were:

1. Implement heterogeneous computing support for agent-based modelling
2. Devise a general-purpose scheme to speed up GPU simulations with oversubscribed memory by using program analysis to statically figure out what memory needed to be loaded ahead of time.

The second goal, in particular, turned out to be both technically infeasible as well as beyond the scope of a thesis project. While such techniques have yielded promising results for problems such as dependency detection and security auditing of programs, there were strong doubts as to whether it was possible to determine the behavior of a program (which varies widely according to input data) at compile time to such an extent that memory access patterns could be accurately deciphered. Resultantly, the project underwent a change in scope and direction in the beginning part of the quarter.

Broadly, these changes in scope may be described as pivoting the project to be focused on providing heterogenous support in MASS/AMB systems which demonstrably:

1. Scales similarly, despite being a little slower for a given problem size, to a pure CUDA heterogenous implementation of a given algorithm.
2. Does not require any knowledge of CUDA or manual memory/resource management on the part of the end user.

Then, primarily, we wish to provide big gains in productivity for running computations on GPUs with a minimal and tolerable loss in performance as compared to using pure CUDA. These updated goals contrast with the initial thesis plan which was focused on improving core heterogeneous computing performance on its own (regardless of whether it was CUDA/MASS etc.)

Updated deliverables and proposed evaluation criteria
In line with the updated thesis plan, we propose the following deliverables and evaluation criteria:

Table 1 Deliverables

<table>
<thead>
<tr>
<th></th>
<th>Fall 2019</th>
<th>Winter 2020</th>
<th>Spring 2020 (if needed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Implement three representative algorithms as Pure CUDA heterogeneous versions and analyze their programmability/performance</td>
<td>Add Heterogenous support to MASS</td>
<td>Finish write up etc.</td>
</tr>
<tr>
<td></td>
<td>Devise strategy to add heterogeneous support to MASS</td>
<td>Benchmark and analyze work in terms of developer productivity, programmability and performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thesis write up</td>
<td></td>
</tr>
</tbody>
</table>

The following questions present a general evaluation criterion to gauge the success of the project:

- **Programmability:** Are there any modifications needed from a normal MASS C++ program to support heterogeneous computing?
- **Performance:** How does the performance scale with pure CUDA heterogeneous algorithms?

Implementing representative heterogeneous algorithms as CUDA programs
To provide a comprehensive benchmark suite and comparison point for MASS-Heterogeneous, we implement three algorithms to explore the facets of programming on GPU’s with oversubscribed memory along qualitative (programmability) as well as quantitative (performance) measures.
The following algorithms were chosen as they represent a good sampling of the type of workloads well suited to MASS:

1. **Heat2D**: A physics simulation of how heat disperses across a 2D surface. Uses Places from MASS
2. **Game of Life**: Utilizes both Agents and Places from MASS as organisms strive to search for sugar in two-dimensional environment.
3. **Neural Network Growth**: The simulation uses Places in a hybrid manner and models the growth of Dendrites and Axons from a Neuron across a two-dimensional surface.

**Heat2D**

Heat2D is a popular physics-based simulation that models the dispersion of heat across a 2D surface.

![Heat2D Simulation](image)

*Figure 1 Modeling heat dispersion across a grid [1]*

To implement the relatively simple Heat2D, we model the 2-dimensional grid as a 1D linearized array (as is the norm in CUDA to avoid deep copying pointers). Our grid is defined by the following three variables:

Initially, heat is applied from the top for a user defined period of time and once this is dissipated we simply model the flow of heat across the grid.

The simulation proceeds with the Euler equation called on all cells in the grid to calculate updated values of temperature for the grid, as shown in Figure 2.
Adding Heterogeneous support

To add heterogeneous support to our Heat 2D simulation, use a scaling factor called strands which breaks our problem space into overlapping subproblems with shared border regions.

In each epoch, a slice is moved to the GPU and its values are updated. It is then merged again into the parent matrix and the process repeats for the next slice.
Sugar Scape

Sugar Scape is a popular ‘game of life’ simulation which first appeared in the 1990’s. While the original author’s aim to model ancient civilizations on it turned out to be too optimistic, it is nonetheless an interesting program and one which is well suited for our benchmark suite.

![Sugar Scape Simulation](image)

**Figure 4** Sugar Scape – the blue dots are organisms and density of green color indicates sugar presence [2]

We model organisms as Agents in MASS while grids containing sugar are modeled as Places. The simulation begins with mountains of Sugar across the grid and randomly assigned agents having randomly assigned metabolisms. Based on the amount of food and pollution present, agents go to different grid points and either feed or die off.

This is a considerably more complex problem to model than Heat 2D due to the amount of data for each grid cell as well as its interdependencies. We follow the CUDA best practice to prefer multiple arrays to model the simulation rather than relying on a single array containing structs:

```c
// Create a space:
double *pollution, *avePollution;

int nBytesInt = sizeof(int) * size * size;
int nBytesDbl = sizeof(double) * size * size;

// Allocate memory on the device:
CATCH (cudaMalloc((void**) &curSugar, nBytesInt));
CATCH (cudaMalloc((void**) &maxSugar, nBytesInt));
CATCH (cudaMalloc((void**) &nAgentsInPlace, nBytesInt));
CATCH (cudaMalloc((void**) &nextAgentIdx, nBytesInt));
CATCH (cudaMalloc((void**) &destinationIdx, nBytesInt));
CATCH (cudaMalloc((void**) &agentSugar, nBytesInt));
CATCH (cudaMalloc((void**) &agentMetabolism, nBytesInt));
CATCH (cudaMalloc((void**) &pollution, nBytesDbl));
CATCH (cudaMalloc((void**) &avePollution, nBytesDbl));
```

**Figure 5** Using arrays to represent different variables per grid cell for Sugar Scape.

The simulation is itself proceeds as follows. For each epoch, we update the information in each of the arrays by calling the following respective kernel functions:
Figure 6 Kernel functions called to update values per epoch

Adding heterogeneous support

While the design paradigm to model the Sugar Scape grid as a collection of arrays simplifies the CUDA implementation, it has the opposite effect when we add heterogeneous support to the application. Due to the large number of data arrays, we must manually partition all the data into overlapping sub problems and then merge it all together. Figure 7 shows the main loop accomplishing this.

Figure 7 The code for breaking sugar scope into overlapping problems and merging the results is well over a hundred lines.

The following figure shows the results of our Sugar Scape execution:
Neural Network Growth

Neural Network Growth models the growth of Neurons, Axons, and Dendrites across a two-dimensional grid as well as the transmittance of signals across them. To simplify our simulation, we focus solely on the growth phase.

![Figure 8 Results of Sugar Scape Execution](image)

Our simulation proceeds as follows:
1. Neurons appear with a random probability on each point on the grid
2. Each Neuron, with a given probability, might produce a single Axon or up to four Dendrites
3. Axons keep growing, with a given probability, either straight ahead or at a 90 degree turn
4. Dendrites keep growing at either a 90 degree turn or straight ahead with a given probability
5. If a Dendrite and Axon meet while facing each other, they ‘fuse’ and stop growing further
6. Each grid can only support a certain number of Axons/Dendrites. Once this maximum is reached the path becomes closed.
7. The simulation continues until either all the Axons have fused with Dendrites or there is no more space left to grow.

Given the difficulty of adding heterogeneous support to Sugarscape, we go against the CUDA best practice of preferring multiple arrays and implement Neural Network Growth as a linearized 2 dimensional array of structs containing pointers.

```c
__device__ int CAPACITE = 15;

struct GridPoint {
    bool hasNeuronBody;
    int neuronID;
    int dendriteCount;

    // Axons and Dendrites. index 0 is reserved for host axon.
    int *guestAxonsDendrites;
};
```

*Figure 10 A grid cell containing a struct with neuron body, axons and dendrites*

Our simulation proceeds with two main kernel functions, formProtrusions and growProtrusions, invoked every epoch. FormProtrusions will probabilistically grow a dendrite/axon from a neuron body (capacity permitting) while growProtrusions grows the dendrites and axons according to our algorithm as space permits.
Due to interdependencies in the way the algorithm is implemented, we must resort to using atomic operations for the growth phase:

```c
if (grid[ i*sizeX + j ].hasNeuronBody && grid[ i*sizeX + j ].guestAxonsDendrites[k] == 0) {
    if (randomFloat() >= probabilityAxon) // make new axon: type = 1, head = 1, dir = rand
        grid[ i*sizeX + j ].guestAxonsDendrites[k] = makeProliferation(grid[ i*sizeX + j ].neuronID, 1);
}
```

**Figure 11** Kernel functions to from Axons and Dendrites

Adding heterogeneous support

Due to our design choice of using just one linearized array for the grid representation, the complexity of adding heterogeneous support is lower than that for Sugarscape. We implement helper functions for abstracting away the deep copying of sections of the array with GridPoint structs and, resultantly, the swap data in and out in a main loop.

```c
if (grid[ new_x*sizeX + new_y ].guestAxonsDendrites[1] == 0) {
    do {
        if (isSet = atomicCAS(mutex, 0, 1) == 0) {
            grid[ new_x*sizeX + new_y ].guestAxonsDendrites[1] = makeProliferation(grid[ i*sizeX + j ].neuronID, 1, directionOfGrowth);
            grid[ i*sizeX + j ].guestAxonsDendrites[k] = setHead(extractIdentifier(proliferation), 0); break;
        }
        if (isSet) mutex = 0;
    } while (!isSet);
}
```

**Figure 12** Utilizing atomic operations for growth
Figure 13 Neural Network Growth: Main loop to split and merge sub problems

Figure 14 shows the results of running our version of Neural Network Growth on a 10x10 grid.

E = no Neuron, N = Neuron, Ax = number of Axons, Dy = number of Dendrites
Performance and Programmability Analysis

Programmability

In general, adding heterogeneous support to a pure CUDA algorithm was found to be a relatively laborious task with intricacies such as precisely managing array/memory copies and pointer arithmetic.

Heat 2D was relatively simple as it was trivially represented by a linearized 2-dimensional array. Neural Network Growth was considerably more complex as deep copies needed to be made for every memory transfer due to the implementation utilizing multiple levels of pointers. Sugar Scape was the most complex to break down as the implementation depended on multiple arrays for representing different facets of the simulation which needed to be manually managed.

Due to the relatively repetitive nature of breaking down and merging overlapping subproblems, the author is of the opinion that a higher-level framework which abstracts away these details for agent-based modeling would be a significant productivity boost to the end user running simulation on oversubscribed GPUs.

Performance

Figure 15 below presents the execution times (in ms) of the algorithms. The results are mostly within expectations as we can see a significant drop in performance as the number of partitions increases. This is due to the fact that DRAM <-> GPU memory data transfers are very slow, relatively speaking, and comprise most of the performance bottleneck.

Heat 2D is using a simple 2D array and is the fastest. Sugarscape uses multiple arrays to represent its data which is probably the cause of its slower result. Neural Network Growth uses deep copies and multiple levels of pointers which explains it being the slowest.

One interesting takeaway is that while a nested pointer structure (as opposed to multiple arrays) made for easier implementation of heterogeneous computing, the deep copying on every epoch incurred a performance penalty.

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>100x100</th>
<th>1000x1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat 2D – 5 partitions</td>
<td>108020</td>
<td>2116701</td>
</tr>
<tr>
<td>Heat 2D – 50 partitions</td>
<td>376129</td>
<td>20175825</td>
</tr>
<tr>
<td>Sugar Scape – 5 partitions</td>
<td>101975</td>
<td>982912</td>
</tr>
<tr>
<td>Sugar Scape – 50 partitions</td>
<td>111811</td>
<td>19183064</td>
</tr>
<tr>
<td>Neural Network – 5 partitions</td>
<td>145216</td>
<td>6342141</td>
</tr>
<tr>
<td>Neural Network – 50 partitions</td>
<td>143782</td>
<td>31232111</td>
</tr>
</tbody>
</table>

*Figure 15 Scalability of heterogeneous versions of algorithms*

Plan for Next quarter

The main emphasis in Winter 2020 will be on adding robust heterogenous computing support to the MASS library.

MASS centers around two central concepts: Places and Agents. To illustrate them, consider a generic problem of connectivity in a directed acylicial graph. To map this problem to MASS, the
most logical mapping would be of the nodes of the graph to Places while Agents would be executable code exploring connectivity on the graph.

Hence, to add heterogenous computing support to MASS, we need to build appropriate support into MASS CUDA for handling Places and Agents while swapping computationally relevant parts of the program into the GPU memory. MASS has the concept of Agents which move between Places as discussed previously. Places.callAll() and manageAll() are methods which must be applied to all places and hence, our strategy relies on partitioning Places into small strips which are moved to the GPU for computation. As exchangeAll needs to access neighbouring places boundaries, we need to include boundary areas from all the neighbours.

Similarly, Agents live on places and need to access the variables from that Place as part of its lifecycle. Hence, as we move sub lists of agents to the GPU, we simultaneously need to move the places these agents need access to as well.

Critically, MASS’ existing design has the concept of partitioning into ranks. For Places, this simply means that places are statically set to places 1…N and these cannot change. Hence, we need to modify PlacesModel to be able to query and selectively offload x amount of partitions onto the GPU.

AgentsModel, on the other hand, can have the number of agents assigned to a rank vary and this necessitates the inclusion of monitoring code to remove ghost agents and decide what ranks to push to the GPU.

Appendix

Citations
2 - http://www.petercollingridge.co.uk/blog/alife/sugarscape/recreation/
3 - https://www.jneurosci.org/content/24/15/3890

Source Code

Neural Network Growth

```c++
#include <thrust/version.h>
#include <iostream>
#include <random>
#include <sstream>
#include <iomanip>
#include <string>
#include <random>
#include <cfloat>
#include <curand.h>
#include <curand_kernel.h>
#include <math.h>

// Helpers
```
__host__ __device__
int extractIdentifier(int input) {
    return input%100000;
}

__host__ __device__
int extractType(int number) {
    return number/10000;
}

__host__ __device__
int isHead(int number) {
    return (number%10000)/100;
}

__host__ __device__
int getDir(int number) {
    return ((number%10000)%100);
}

__host__ __device__
int setType(int number, int type) {
    int currType = extractType(number);
    if (currType == 1 && type == 2) number += 1000;
    if (currType == 2 && type == 1) number -= 1000;
    return number;
}

__host__ __device__
int setHead(int number, int type) {
    int currType = isHead(number);
    if (currType == 0 && type == 1) number += 100;
    if (currType == 1 && type == 0) number -= 100;
    return number;
}

// 1 means: x+1, 2 means: x-1, 3 means: y+1, 4 means: y-1
__host__ __device__
int setDir(int number, int type) {
    int currType = getDir(number);
    return number-currType+type;
}

__host__ __device__
bool isOppositeDir(int dir, int dir2) {
    return (dir == 1 && dir2 == 2) || (dir == 3 && dir2 == 4);
}

std::random_device rd;
std::mt19937 rand_engine(rd());
std::uniform_real_distribution<> uniform_zero_to_one(0.0, 1.0);
std::uniform_real_distribution<> uniform_one_to_four(1.0, std::nextafter(4.0, DBL_MAX));
std::uniform_real_distribution<> uniform_one_to_three(0.0, std::nextafter(3.0, DBL_MAX));

__device__ int randomBetween(const unsigned max_rand_int, const unsigned min_rand_int){
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int seed = idx;
curandState s // seed a random number generator

curand_init(seed, 0, 0, &s);

float myrandf = curand_uniform(&s);
myrandf *= (max_rand_int - min_rand_int + 0.999999);
myrandf += min_rand_int;
int myrand = (int)truncf(myrandf);
//return myrand;

return ceilf(curand_uniform(&s) * max_rand_int);
}

__device__ float randomFloat() {
    int idx = threadIdx.x + blockDim.x*blockIdx.x;
    unsigned int seed = idx;
    curandState s // seed a random number generator
    curand_init(seed, 0, 0, &s);
    float myrandf = curand_uniform(&s);
    return myrandf;
}

__device__ int makeProtrusion(int neuronID, int type) {
    int identifier = setHead(type * 10000, 1);
    identifier = setDir(identifier, randomBetween(4, 1));
    int times = 1;
    while (times <= identifier)
        times *= 10;
    return neuronID*times + identifier;
}

__device__ int makeProtrusion(int neuronID, int type, int dir) {
    int identifier = setHead(type * 10000, 1);
    identifier = setDir(identifier, dir);
    int times = 1;
    while (times <= identifier)
        times *= 10;
    return neuronID*times + identifier;
}

__device__ int getGrowthDir(int dir) {
    int dirs[3];
    if (dir == 1) {dirs[0] = 1; dirs[1] = 4; dirs[2] = 3;}
    if (dir == 2) {dirs[0] = 2; dirs[1] = 4; dirs[2] = 3;}
    if (dir == 3) {dirs[0] = 3; dirs[1] = 1; dirs[2] = 2;}
    if (dir == 4) {dirs[0] = 4; dirs[1] = 1; dirs[2] = 2;}
    int res = randomBetween(1, 3);
    return dirs[res];
\begin{verbatim}

int CAPACITY = 15;
__device__ int CAPACITE = 15;

struct GridPoint {
    bool hasNeuronBody;
    int neuronID;
    int dendriteCount;

    // Axons and Denrites. index 0 is reserved for host axon.
    int *guestAxonsDendrites;
};

void makeGrid(GridPoint *array, int sizeX, int sizeY, float chanceOfNeuron) {
    int neuronID = 0;
    for (int i = 0; i < sizeX*sizeY; i++) {
        if (uniform_zero_to_one(rand_engine) >= chanceOfNeuron) {
            array[i].hasNeuronBody = true;
            array[i].neuronID = neuronID;
            neuronID++;
        } else {
            array[i].hasNeuronBody = false;
        }
        array[i].dendriteCount = 0;
        array[i].guestAxonsDendrites = new int[CAPACITY];
        for (int j = 0; j < CAPACITY; j++)
            array[i].guestAxonsDendrites[j] = 0;
    }
}

void printGrid(GridPoint *array, int sizeX, int sizeY) {
    for (int i = 0; i < sizeX*sizeY; i++) {
        std::cout << array[i].hasNeuronBody << "|" << array[i].neuronID << "| Axons and Denrites:"
    }
    for (int j = 0; j < CAPACITY; j++)
        std::cout << array[i].guestAxonsDendrites[j] << ", "
    std::cout << "\n"
}

void printMatrix(GridPoint *grid, int sizeX, int sizeY) {
    for (int i = 0; i < sizeX; i++) {
        for (int j = 0; j < sizeY; j++) {
            std::string output = ""
            if (grid[ i*sizeX + j ].hasNeuronBody == 1)
                output += "N";
            else
                output += "E";

\end{verbatim}
int axonCount = 0;
int dendriteCount = 0;
for (int k = 0; k < CAPACITY; k++) {
  int protrusion = grid[ i*sizeX + j ].guestAxonsDendrites[k];
  if (protrusion != 0) {
    int type = extractType(extractIdentifier(protrusion));
    if (type == 1) axonCount += 1;
    if (type == 2) dendriteCount += 1;
  }
}
output += "A" + std::to_string(axonCount);
output += "D" + std::to_string(dendriteCount);
std::stringstream ss;
ss << std::setw(5) << output;
std::cout << ss.str() << " ";
std::cout << "\n";
}

__global__ void formProtrusions(GridPoint* grid, int sizeX, int sizeY, float probabilityAxon, float probabilityDendrite) {
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int *h_tmp = new int[CAPACITY];
  cudaMemcpy(h_tmp, temp[i].guestAxonsDendrites, sizeof(int) * CAPACITY, cudaMemcpyDeviceToHost);
  temp[i].guestAxonsDendrites = h_tmp;
}

void copyGridToDevice(GridPoint *array, GridPoint *d_a, int sizeX, int sizeY) {
  GridPoint *temp = new GridPoint[sizeX*sizeY];
  for (int i = 0; i < sizeX*sizeY; i++) {
    temp[i].hasNeuronBody = array[i].hasNeuronBody;
    temp[i].neuronID = array[i].neuronID;
    temp[i].guestAxonsDendrites = new int[CAPACITY];
    int *d_tmp;
    cudaMemcpy(&d_tmp, sizeof(int) * CAPACITY);
    cudaMemcpy(d_tmp, array[i].guestAxonsDendrites, sizeof(int) * CAPACITY, cudaMemcpyHostToDevice);
    temp[i].guestAxonsDendrites = d_tmp;
  }
  cudaMemcpy(d_a, temp, sizeof(GridPoint) * sizeX*sizeY, cudaMemcpyHostToDevice);
}

void copyGridFromDevice(GridPoint *d_a, GridPoint *temp, int sizeX, int sizeY) {
  cudaMemcpy(temp, d_a, sizeof(GridPoint) * sizeX*sizeY, cudaMemcpyDeviceToHost);
  for (int i = 0; i < sizeX*sizeY; i++) {
    int *h_tmp = new int[CAPACITY];
    cudaMemcpy(h_tmp, temp[i].guestAxonsDendrites, sizeof(int) * CAPACITY, cudaMemcpyDeviceToHost);
    temp[i].guestAxonsDendrites = h_tmp;
  }
}
if(i>=sizeX-1 || j>=sizeY-1 || i == 0 || j == 0) return;
if (grid[i*sizeX + j].hasNeuronBody && grid[i*sizeX + j].guestAxonsDendrites[0] == 0)
if (randomFloat() >= probabilityAxon) // make new axon: type = 1, head = 1, dir = rand
    grid[i*sizeX + j].guestAxonsDendrites[0] = makeProtrusion(grid[i*sizeX + j].neuronID, 1);

if (grid[i*sizeX + j].hasNeuronBody && grid[i*sizeX + j].guestAxonsDendrites[0] == 0)
if (randomFloat() >= probabilityDendrite)
    for (int k = 1; k < CAPACITE; k++)
        if (grid[i*sizeX + j].dendriteCount < 4 && grid[i*sizeX + j].guestAxonsDendrites[k] == 0) {
            grid[i*sizeX + j].dendriteCount++;
            grid[i*sizeX + j].guestAxonsDendrites[k] = makeProtrusion(grid[i*sizeX + j].neuronID, 2);
            break;
        }

__global__
void growProtrusions(GridPoint* grid, int sizeX, int sizeY, float probabilityAxon, float probabilityDendrite, unsigned long long* mutex) {
    // Axons
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i>=sizeX-1 || j>=sizeY-1 || i == 0 || j == 0) return;
    //for (int i = 1; i < sizeX-1; i++)
    //  for (int j = 1; j < sizeX-1; j++) {
    for (int k = 0; k < CAPACITE; k++) {
        int protrusion = grid[i*sizeX + j].guestAxonsDendrites[k];
        bool isSet = false;
        if (protrusion != 0 && extractType(extractIdentifier(protrusion)) == 1) {
            if (isHead(extractIdentifier(protrusion)) == 1) {
                int dir = getDir(extractIdentifier(protrusion));
                // check if opposing exists
                bool oppositeExists = false;
                for (int l = 1; l < CAPACITE; l++) {
                    int test = grid[i*sizeX + j].guestAxonsDendrites[l];
                    if (extractType(extractIdentifier(test)) == 2 // checks for fusion on every epoch, rather than freezing, potential performance degradation
                        && isHead(extractIdentifier(test)) == 1
                        && isOppositeDir(getDir(extractIdentifier(test)), dir)) oppositeExists = true;
                }
                // grow if space available
                if (!oppositeExists) {
                    int directionOfGrowth = getGrowthDir(dir);
                    int new_x = i;
                    int new_y = j;
                    if (directionOfGrowth == 1) new_x += 1;
                    if (directionOfGrowth == 2) new_x -= 1;
                    if (directionOfGrowth == 3) new_y += 1;
                    if (directionOfGrowth == 4) new_y -= 1;
                    for (int l = 1; l < CAPACITE; l++)
                        if (grid[new_x*sizeX + new_y].guestAxonsDendrites[l] == 0) {
                            do {
                                if (isSet = atomicCAS(mutex, 0, 1) == 0) {
                                    
                                }
                            } while (isSet = atomicCAS(mutex, 0, 1) == 0) {
                        }
                    }
                }
            }
        }
    }
}

// Dendrites

// Fusion

// Axon and Dendrite Fusion

// Electrical Synapse

// Chemical Synapse

// Gap Junction

// Neurotransmitter

// Neurotransmitter Receptor

// Neurotransmitter Synthesis

// Neurotransmitter Degradation

// Neurotransmitter Transport

// Neurotransmitter Release

// Neurotransmitter Reuptake

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neurotransmitter Release Inhibitor

// Neurotransmitter Release Activator

// Neurotransmitter Synthesis Inhibitor

// Neurotransmitter Synthesis Activator

// Neurotransmitter Reuptake Inhibitor

// Neurotransmitter Reuptake Activator

// Neurotransmitter Transport Inhibitor

// Neurotransmitter Transport Activator

// Neur
grid[ new_x*sizeX + new_y ].guestAxonsDendrites[l] = makeProtrusion(grid[ i*sizeX + j ].neuronID, 1, directionOfGrowth);

grid[ i*sizeX + j ].guestAxonsDendrites[k] = setHead(extractIdentifier(protrusion), 0);
break;
}

if (isSet)
{
mutex = 0;
}
}
while (!isSet);

}

}

}

}

if (protrusion != 0 && extractType(extractIdentifier(protrusion)) == 2) {
if (isHead(extractIdentifier(protrusion)) == 1) {
int dir = getDir(extractIdentifier(protrusion));
// check if opposing exists
bool oppositeExists = false;
int test = grid[ i*sizeX + j ].guestAxonsDendrites[0]; // axon is always at index 0
if (isHead(extractIdentifier(test)) == 1
&& isOppositeDir(getDir(extractIdentifier(test)), dir) oppositeExists = true;
// grow if space available
if (!oppositeExists) {
int directionOfGrowth = getGrowthDir(dir);
int new_x = i;
int new_y = j;
if (directionOfGrowth == 1) new_x += 1;
if (directionOfGrowth == 2) new_x -= 1;
if (directionOfGrowth == 3) new_y += 1;
if (directionOfGrowth == 4) new_y -= 1;
for (int l = 1; l < CAPACITE; l++)
if (grid[ new_x*sizeX + new_y ].guestAxonsDendrites[l] == 0) {
do

    if (isSet = atomicCAS(mutex, 0, 1) == 0)
    {
        grid[ new_x*sizeX + new_y ].guestAxonsDendrites[l] = makeProtrusion(grid[ i*sizeX + j ].neuronID, 2, directionOfGrowth);
        grid[ i*sizeX + j ].guestAxonsDendrites[k] = setHead(extractIdentifier(protrusion), 0);
break;
    }
if (isSet)
{
    mutex = 0;
}
}
while (!isSet);
}
}
//
}
// Synchronisation issues
// partition issues

void makeCopyToDevice() {
    // allocate on host
    int sizeX = 32; // 0 10, 10 20, 20 30, 25 32
    int sizeY = 32;
    GridPoint *array = new GridPoint[sizeX*sizeY];

    makeGrid(array, sizeX, sizeY, 0.5);

    //printMatrix(array, sizeX, sizeY);

    GridPoint *arr2 = new GridPoint[8*32];

    for (int ii = 0; ii < 1000; ii++) {
        for (int i = 0; i < 32*32; i+=8*32) {
            for (int j = 0; j < 8*32+1; j++) {
                if (j+i < 32*32) {
                    arr2[j] = array[j + i];
                }
            }
        }
        GridPoint *d_a;
        cudaMalloc(&d_a, sizeof(GridPoint) * 8 * 32);
        copyGridToDevice(array, d_a, 8, 32);
        cudaDeviceSynchronize();
    }

    // Threads per CTA dimension
    int THREADS = 32;

    // Blocks per grid dimension (assumes THREADS divides N evenly)
    int BLOCKS = (8*32) / THREADS;

    // Use dim3 structs for block and grid dimensions
    dim3 threads(THREADS, THREADS);
    dim3 blocks(BLOCKS, BLOCKS);

    // for (int i = 0; i < 20; i++) {
    // formProtrusions<<<blocks, threads>>>(d_a, 8, 32, 0.3, 0.25);
    // cudaDeviceSynchronize();
    // }
    // for (int i = 0; i < 10; i++) {
    // growProtrusions<<<blocks, threads>>>(d_a, 8, 32, 0.2, 0.1, 0);
    // cudaDeviceSynchronize();
    // }

    GridPoint* h_a = new GridPoint[8*32];
    copyGridFromDevice(d_a, h_a, 8, 32);
    printMatrix(h_a, 8, 32);

    for (int j = 0; j < 8*32+1; j++) {
        if (j+i < 32*32) {
            array[i + j] = h_a[j];
        }
    }
// Begin implementation of custom CUDA implementation of HEAT2D

__device__ int getGlobalIdx_1D_1D() {
    return blockIdx.x * blockDim.x + threadIdx.x;
}

__global__ void setCold(double *dest, double *src, int size) {
    int idx = getGlobalIdx_1D_1D();
    if (idx < size * size) {
        dest[idx] = 0.0;
        src[idx] = 0.0;
    }
}
__device__ bool isLeftEdge(int idx, int size) {
    return idx % size == 0;
}

__device__ bool isRightEdge(int idx, int size) {
    return idx % size == size - 1;
}

__device__ bool isTopEdge(int idx, int size) {
    return idx < size;
}

__device__ bool isBottomEdge(int idx, int size) {
    return idx >= size * size - size;
}

__global__ void setEdges(double *dest, double *src, int size, int t,
                          int heat_time, double r) {
    int idx = getGlobalIdx_1D_1D();
    if (idx < size * size) {
        // apply heat to top row
        if (idx >= 0 && idx < size && t < heat_time) {
            src[idx] = 19.0; // heat
            return;
        }
        // two left-most and two right-most columns are identical
        if (isLeftEdge(idx, size)) {
            src[idx] = src[idx + 1];
            return;
        }
        if (isRightEdge(idx, size)) {
            src[idx] = src[idx - 1];
            return;
        }
        // two upper and lower rows are identical
        if (isTopEdge(idx, size)) {
            src[idx] = src[idx + size];
            return;
        }
        if (isBottomEdge(idx, size)) {
            src[idx] = src[idx - size];
            return;
        }
    }
}

__global__ void euler(double *dest, double *src, int size, int t, int heat_time,
                      double r) {
    int blockIdx = blockIdx.x + blockIdx.y * gridDim.x;
    int threadIdx = blockIdx * (gridDim.x * blockDim.y) + threadIdx.y * blockDim.x + threadIdx.x;
int idx = threadId;

// perform forward Euler method
if (idx > size && idx < (size * size - size)) {
  double tmp = src[idx];
  dest[idx] = tmp + r * (src[idx + 1] - 2 * tmp + src[idx - 1])
    + r * (src[idx + size] - 2 * tmp + src[idx - size]);
}
}

void copyToDevice(double *src, double *dest, int size) {
}

void printMat(vector<double> linMat, int sizeX, int sizeY) {
  for (int i = 0; i < sizeX; i++) {
    for (int j = 0; j < sizeY; j++) {
      cout << linMat[i * sizeX + j] << "\t";
    }
    cout << "\n";
  }
  cout << "\n\n\n";
}

void runDeviceSim(int size, int max_time, int heat_time, int interval, int strands) {
  double r = a * dt / (dd * dd);

  // CUDA code here
  double *dest, *src;
  int nBytes = sizeof(double) * size * size;
  cudaMalloc((void**) &dest, nBytes);
  cudaMalloc((void**) &src, nBytes);
  int gridWidth = (size * size - 1) / WORK_SIZE + 1;
  int threadWidth = (size * size - 1) / gridWidth + 1;
  dim3 gridDim(gridWidth);
  dim3 threadDim(threadWidth);
  vector<double> original(size*size*strands, 0.0);
  printMat(original, size, size*strands);
  for (int t = 0; t < max_time; t++) {
    int prev = 0;
    int bottom = 0;
    for (int i = size*size-1; i <= size*size*strands; i += size*size-size-1) {
      if (prev != 0) bottom = prev - size;
      std::vector<double> slice(size * size);
      copy(original.begin()+bottom, original.begin()+i, slice.begin());
      cudaMemcpy(src, slice.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
      if (i >= (size*size*strands) / 3 && i < (size*size*strands) / 3 * 2 && t < heat_time) {
        setEdges<<<gridDim, threadDim>>>(dest, src, size, t, heat_time, r);
      }
      cudaMemcpy(dest, src, size*size * sizeof(int), cudaMemcpyDeviceToHost);
    }
  }
}

}  
euler<<<gridDim, threadDim>>>(dest, src, size, t, heat_time, r);  
cudaDeviceSynchronize();

    double *swap = dest;
    dest = src;
    src = swap;

    cudaMemcpy(slice.data(), src, size*size * sizeof(int), cudaMemcpyDeviceToHost);
    // printMat(slice, size, size);
    copy(slice.begin(), slice.end(), original.begin()+bottom);
    cout << bottom << " " << i << " " << i-bottom << endl;
    prev = i;
}  
//printMat(original, size, size*strands);
}

#include <chrono>
using namespace std::chrono;

int main() {
    // int size[10] = { 5, 10, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
    int max_time = 10;
    int heat_time = 5;
    int interval = 1;
    int strands = 5;
    auto start = high_resolution_clock::now();

    runDeviceSim(900, max_time, heat_time, interval, strands);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "End of the GPU simulation" << duration.count() << endl;

    return 0;
}

SugarScape

%%%cu

#include <ctime>  // clock_t
#include <stdio.h>
#include <iostream>
#include <math.h>  // floor
#include <sstream>  // ostringstream
#include <vector>
#include <cuda_runtime.h>
#include <curand.h>  // for CUDA’s random number stuff
#include <curand_kernel.h>  // for CUDA’s random number stuff
#include <vector>
#define CERR
#define CATCH(err) __cudaCatch( err, __FILE__, __LINE__ )
#define CHECK() __cudaCheckError( __FILE__, __LINE__ )

static const int WORK_SIZE = 32; //num of threads per block
static const int HALO_SPACE = 1; //number of cells on each side required for calculations
static const int maxVisible = 3;
static const int maxMetabolism = 4;
static const int maxInitAgentSugar = 10;
static const bool show = false; //print progress from gpu

using namespace std;
int initSugarAmount(int idx, int size, int sizeY, int mtPeakX, int mtPeakY, int maxMtSug);  

// Sets initial amount of sugar for the cell
void setSugar(int* curSugar, int* maxSugar, int*nextAgentIdx, int*destinationIdx, double*pollution, double*avePollution, int size, int sizeY, int maxMtSug) {
  for (int idx = 0; idx < size*sizeY; idx++) {
    pollution[idx] = 0.0; // the current pollution
    avePollution[idx] = 0.0; // averaging four neighbors' pollution
    destinationIdx[idx] = -1; // the next place to migrate to
    nextAgentIdx[idx] = -1;  // the next agent to come here
    int mtCoord[2];
    mtCoord[0] = size/3;
    mtCoord[1] = size - size/3 - 1;

    int mt1 = initSugarAmount(idx, size, sizeY, mtCoord[0], mtCoord[1], maxMtSug);
    int mt2 = initSugarAmount(idx, size, sizeY, mtCoord[1], mtCoord[0], maxMtSug);
    curSugar[idx] = mt1 > mt2 ? mt1 : mt2;
    maxSugar[idx] = mt1 > mt2 ? mt1 : mt2;
  }
}

int initSugarAmount(int idx, int size, int sizeY, int mtPeakX, int mtPeakY, int maxMtSug) { // TODO: smooth our
  int x_coord = idx % size;
  int y_coord = idx / sizeY;

  double distance = sqrt((float)(( mtPeakX - x_coord ) * ( mtPeakX - x_coord ) + (mtPeakY - y_coord) * 
                             (mtPeakY - y_coord)));

  // radius is assumed to be simSize/2.
  int r = size/2;
  if ( distance < r )
  {
    // '+ 0.5' for rounding a value.
    return ( int )( maxMtSug + 0.5 - maxMtSug / ( double )r * distance );
  }
  else
    return 0;
}

void initAgents(int*nAgentsInPlace, int nAgents, int size, int sizeY, int *agentSugar, int *agentMetabolism) {
  srand (239); //seed
int ratio = 100 * nAgents / (size * sizeY); // percentage of non-empty cells

for (int idx = 0; idx < size * sizeY; idx++) {
    unsigned int randNumber = rand() % 100; // random number from 0 to 100
    if (randNumber <= ratio) {
        nAgentsInPlace[idx] = 1;
        agentSugar[idx] = rand() % maxInitAgentSugar + 1;
        agentMetabolism[idx] = rand() % maxMetabolism + 1;
    } else {
        nAgentsInPlace[idx] = 0;
        agentSugar[idx] = -1;
        agentMetabolism[idx] = -1;
    }
}

// __device__ int initSugarAmount(int idx, int size, int mtPeakX, int mtPeakY, int maxMtSug);

// Utility functions:
__device__ int getGlobalIdx_1D_1D() {
    return blockIdx.x * blockDim.x + threadIdx.x;
}

void __cudaCatch(cudaError err, const char *file, const int line) {
    #ifdef CERR
    if (cudaSuccess != err) {
        cerr << "MASS Cuda Util: " << cudaGetErrorString(err) << " in " << file << " at line " << line << endl;
        exit(EXIT_FAILURE);
    }
    #endif
}

void __cudaCheckError(const char *file, const int line) {
    #ifdef CERR
    __cudaCatch(cudaGetLastError(), file, line);
    #endif
}

// Actual simulation functions:
// Sets initial amount of sugar for the cell
__global__ void setSugar(int *curSugar, int *maxSugar, int *nextAgentIdx, int *destinationIdx,
    double *pollution, double *avePollution, int size, int maxMtSug) {
    int idx = getGlobalIdx_1D_1D();

    pollution[idx] = 0.0; // the current pollution
    avePollution[idx] = 0.0; // averaging four neighbors' pollution
    destinationIdx[idx] = -1; // the next place to migrate to
    nextAgentIdx[idx] = -1; // the next agent to come here
int mtCoord[2];
mtCoord[0] = size/3;
mtCoord[1] = size - size/3 - 1;

int mt1 = initSugarAmount(idx, size, mtCoord[0], mtCoord[1], maxMtSug);
int mt2 = initSugarAmount(idx, size, mtCoord[1], mtCoord[0], maxMtSug);

curSugar[idx] = mt1 > mt2 ? mt1 : mt2;
maxSugar[idx] = mt1 > mt2 ? mt1 : mt2;
}

__device__ int initSugarAmount(int idx, int size, int mtPeakX, int mtPeakY, int maxMtSug) {
    int x_coord = idx % size;
    int y_coord = idx / size;

    double distance = sqrt((float)((mtPeakX - x_coord) * (mtPeakX - x_coord) + (mtPeakY - y_coord) * (mtPeakY - y_coord)));
    // radius is assumed to be simSize/2.
    int r = size/2;
    if (distance < r)
    {
        // '+ 0.5' for rounding a value.
        return (int)(maxMtSug + 0.5 - maxMtSug / (double)r * distance);
    }
    else
    return 0;
}

void initAgents(dim3 gridDim, dim3 threadDim, int *nAgentsInPlace, int nAgents, int size, int *agentSugar, int *agentMetabolism)
{
    // randomly allocate agents on CPU:
    int *nAgentsInPlace_host = new int[size*size];
    int *agentSugar_host = new int[size*size];
    int *agentMetabolism_host = new int[size*size];

    srand (239); //seed
    int ratio = 100 * nAgents / (size*size);  //percentage of non-empty cells

    for (int i = 0; i < size*size; i++) {
unsigned int randNumber = rand() % 100; //random number from 0 to 100
    if (randNumber <= ratio) {
        nAgentsInPlace_host[i] = 1;
        agentSugar_host[i] = rand() % maxInitAgentSugar + 1;
        agentMetabolism_host [i] = rand() % maxMetabolism + 1;
    }
    else
    nAgentsInPlace_host[i] = 0;
    agentSugar_host[i] = -1;
    agentMetabolism_host [i] = -1;
    }

    //Copy the matrix to the host:
    cudaError_t cudaStat = cudaMemcpy(nAgentsInPlace, nAgentsInPlace_host, size*size * sizeof(int),
cudaMemcpyHostToDevice);
  if (cudaStat != cudaSuccess) {
    cerr << "cudaMemcpy failed!" << endl;
    exit(EXIT_FAILURE);
  }
  cudaStat = cudaMemcpy(agentSugar, agentSugar_host, size*size * sizeof(int), cudaMemcpyHostToDevice);
  if (cudaStat != cudaSuccess) {
    cerr << "cudaMemcpy failed!" << endl;
    exit(EXIT_FAILURE);
  }
  cudaStat = cudaMemcpy(agentMetabolism, agentMetabolism_host, size*size * sizeof(int), cudaMemcpyHostToDevice);
  if (cudaStat != cudaSuccess) {
    cerr << "cudaMemcpy failed!" << endl;
    exit(EXIT_FAILURE);
  }
  delete nAgentsInPlace_host;
  delete agentSugar_host;
  delete agentMetabolism_host;
}
*/

__global__ void incSugarAndPollution(int *curSugar, int *maxSugar, double *pollution) {
  int idx = getGlobalIdx_1D_1D();
  if (curSugar[idx] < maxSugar[idx]) {
    curSugar[idx]++;
    pollution[idx] += 1.0;
  }
}

// Calculates average pollution between 4 neighbors(top, bottom, left, right)
__global__ void avePollutions(double *pollution, double *avePollution, int size) {
  int idx = getGlobalIdx_1D_1D();
  if ((idx < size*size) {
    // Using stencil pattern to optimize for the use of shared memory.
    // At this point doing as 1D model and cache left and right halo, but not top and bottom.
    __shared__ double pollution_cached[WORK_SIZE+2*HALO_SPACE];
    //printf("pollution_cached[%d] = pollution[%d]\n", HALO_SPACE+idx%WORK_SIZE, idx);
    pollution_cached[HALO_SPACE+idx%WORK_SIZE] = pollution[idx];
    for (int i=0; i<HALO_SPACE; i++) {
      int index_left = idx/WORK_SIZE * WORK_SIZE -HALO_SPACE +i;
      if (index_left >= 0) {
        //printf("Thread idx = %d, left halo index to be cached= %d\n", idx, index_left);
        pollution Cached[i] = pollution[index_left];
      }
      int index_right = idx/WORK_SIZE * WORK_SIZE + WORK_SIZE +i;
      if (index_right < size*size) {
        //printf("Thread idx = %d, right halo index to be cached= %d, cell to be assigned to = %d\n", idx, index_right, HALO_SPACE + WORK_SIZE + i);
        pollution Cached[HALO_SPACE + WORK_SIZE + i] = pollution[index_right];
      }
    }
__syncthreads();
//printf("Past __syncthreads for thread %d\n", idx);

double top, bottom, right, left;
idx + size < size*size ? top = pollution[idx + size] : 0.0;
idx - size >= 0 ? bottom = pollution[idx-size] : 0.0;

//printf("right pollution_cached[%d]\n", idx%WORK_SIZE + 1);
(idx +1) % size != 0 ? right = pollution_cached[HALO_SPACE + idx%WORK_SIZE + 1] : 0.0; //from shared memory

//printf("left pollution_cached[%d]\n", idx%WORK_SIZE -1);
(idx -1) % size != size-2 ? left = pollution_cached[HALO_SPACE + idx%WORK_SIZE -1] : 0.0; //from shared memory

avePollution[idx] = ( top + bottom + left + right ) / 4.0;
}

__global__ void updatePollutionWithAverage(double *pollution, double *avePollution) {
int idx = getGlobalIdx_1D_1D();
pollution[idx] = avePollution[idx];
avePollution[idx] = 0.0;
}

__global__ void findPlaceForMigration(int *nAgentsInPlace, int *curSugar, double *pollution, int maxVisible, int size, int *destinationIdx)
{
int idx = getGlobalIdx_1D_1D();

if (nAgentsInPlace[idx] > 0) {
    // consider visibility along x axis:
    for (int i = 1; i<=maxVisible; i++) {
        int destIdx = idx+i;
        if ((destIdx < size*size) && nAgentsInPlace[destIdx] == 0 &&
            (curSugar[destIdx] / ( 1.0 + pollution[destIdx]) > 0.0)) {
            destinationIdx[idx] = destIdx;
            return;
        }
    }

    // consider visibility along y axis:
    for (int i = 1; i<=maxVisible; i++) {
        int destIdx = idx+i*size;
        if ((destIdx < size*size) && nAgentsInPlace[destIdx] == 0 &&
            (curSugar[destIdx] / ( 1.0 + pollution[destIdx]) > 0.0)) {
            destinationIdx[idx] = destIdx;
            return;
        }
    }
}
}


```c
__global__ void selectAgentToAccept(int *destinationIdx, int *nextAgentIdx) {
    int idx = getGlobalIdx_1D_1D();
    int destIdx = destinationIdx[idx];

    atomicCAS(nextAgentIdx + destIdx, -1, idx); // atomic operation to assign agent to a free place
    atomicMin(nextAgentIdx + destIdx, idx); // atomic operation to replace the agent in a place if it's idx is smaller than existing
}

__global__ void migrate(int *nAgentsInPlace, int *destinationIdx, int *nextAgentIdx, int *agentSugar, int *agentMetabolism) {
    int idx = getGlobalIdx_1D_1D();
    if (nextAgentIdx[destinationIdx[idx]] == idx) {
        nAgentsInPlace[destinationIdx[idx]] = 1;
        nAgentsInPlace[idx] = 0;
        // move whatever data agents have:
        agentSugar[destinationIdx[idx]] = agentSugar[idx];
        agentSugar[idx] = -1;

        agentMetabolism[destinationIdx[idx]] = agentMetabolism[idx];
        agentMetabolism[idx] = -1;
    }
}

__global__ void resetMigrationData (int *destinationIdx, int *nextAgentIdx) {
    int idx = getGlobalIdx_1D_1D();
    destinationIdx[idx] = -1; // the next place to migrate to
    nextAgentIdx[idx] = -1; // the next agent to come here
}

__global__ void metabolize(int *nAgentsInPlace, int *agentSugar, int *agentMetabolism, int *curSugar, double *pollution) {
    int idx = getGlobalIdx_1D_1D();
    if (nAgentsInPlace[idx] > 0) {
        agentSugar[idx] += curSugar[idx];
        agentSugar[idx] -= agentMetabolism[idx];

        curSugar[idx] = 0;
        pollution[idx] += agentMetabolism[idx];

        if( agentSugar[idx] < 0 )
            // Kill agent:
            nAgentsInPlace[idx] = 0;
            agentSugar[idx] = -1;
            agentMetabolism[idx] = -1;
    }
}

void displayResult(int *results_matrix_h, int sizeX, int sizeY) {
```
for (int i=0; i<sizeX; i++) {
    for (int j=0; j<sizeY; j++) {
        cout << results_matrix_h[i*sizeX+j] << "\t";
    }
    cout << endl;
}
}

void displayResults(int *results_matrix, int size) {
    int *results_matrix_h = new int[size*size];

    //Copy the matrix to the host:
    cudaMemcpy(results_matrix_h, results_matrix, size*size * sizeof(int), cudaMemcpyDeviceToHost));

    for (int i=0; i<size; i++) {
        for (int j=0; j<size; j++) {
            cout << results_matrix_h[i*size+j] << "\t";
        }
        cout << endl;
    }
    delete results_matrix_h;
}

void runDeviceSim(int size, int max_time, int nAgents, int strands) {
    cudaMemcpy_t cudaMemcpy;

    int device_Count;
    cudaMemcpy(device_Count);
    cout << "Device Numbers = " << device_Count << endl;

    cudaMemcpy cudaMemcpy(0);
    cudaMemcpy cudaMemcpy(cudaSuccess) {
        cerr << "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?" << cudaMemcpy << endl;
        exit(EXIT_FAILURE);
    }

    int maxMtSugar = 4; //max level of sugar in mountain peak

    // vectors
    std::vector<int> ovCurSugar(size*size*strands);
    std::vector<int> ovMaxSugar(size*size*strands);

    std::vector<int> ovAgentsInPlace(size*size*strands);
    std::vector<int> ovAgentSugar(size*size*strands);
    std::vector<int> ovAgentMetabolism(size*size*strands);

    std::vector<int> ovDestinationIdx(size*size*strands);
    std::vector<int> ovNextAgentIdx(size*size*strands);

    std::vector<double> ovPollution(size*size*strands);
std::vector<double> ovAvePollution(size*size*strands);

setSugar(ovCurSugar.data(), ovMaxSugar.data(), ovNextAgentIdx.data(), ovDestinationIdx.data(),
        ovPollution.data(), ovAvePollution.data(),
        size, size*strands, maxMtSugar);
initAgents(ovNAgentsInPlace.data(), nAgents, size, size*strands, ovAgentSugar.data(),
        ovAgentMetabolism.data());

// Create a space:
double *pollution, *avePollution;

int nBytesInt = sizeof(int) * size * size;
int nBytesDbl = sizeof(double) * size * size;

// Allocate memory on the device:
CATCH(cudaMalloc((void**) &curSugar, nBytesInt));
CATCH(cudaMalloc((void**) &maxSugar, nBytesInt));
CATCH(cudaMalloc((void**) &nAgentsInPlace, nBytesInt));
CATCH(cudaMalloc((void**) &nextAgentIdx, nBytesInt));
CATCH(cudaMalloc((void**) &destinationIdx, nBytesInt));
CATCH(cudaMalloc((void**) &agentSugar, nBytesInt));
CATCH(cudaMalloc((void**) &agentMetabolism, nBytesInt));
CATCH(cudaMalloc((void**) &pollution, nBytesDbl));
CATCH(cudaMalloc((void**) &avePollution, nBytesDbl));

int gridWidth = (size * size - 1) / WORK_SIZE + 1;
int threadWidth = (size * size - 1) / gridWidth + 1;
dim3 gridDim(gridWidth);
dim3 threadDim(threadWidth);

for (int t = 0; t < max_time; t++) {
    // swapping
    int prev = 0;
    int bottom = 0;
    for (int i = size*size-1; i <= size*size*strands; i += size*size-size-1) {
        if (prev != 0) bottom = prev-size;
        std::vector<int> tmpCurSugar(size*size);
        std::vector<int> tmpMaxSugar(size*size);
        std::vector<int> tmpNAgentsInPlace(size*size);
        std::vector<int> tmpAgentSugar(size*size);
        std::vector<int> tmpAgentMetabolism(size*size);
        std::vector<int> tmpNextAgentIdx(size*size);
        std::vector<double> tmpPollution(size*size);
        std::vector<double> tmpAvePollution(size*size);

        copy(ovCurSugar.begin()+bottom, ovCurSugar.begin()+i, tmpCurSugar.begin());
        copy(ovMaxSugar.begin()+bottom, ovMaxSugar.begin()+i, tmpMaxSugar.begin());
        copy(ovNAgentsInPlace.begin()+bottom, ovNAgentsInPlace.begin()+i, tmpNAgentsInPlace.begin());
        copy(ovAgentSugar.begin()+bottom, ovAgentSugar.begin()+i, tmpAgentSugar.begin());
        copy(ovAgentMetabolism.begin()+bottom, ovAgentMetabolism.begin()+i, tmpAgentMetabolism.begin());
    }
}
cudaMemcpy(curSugar, tmpCurSugar.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(maxSugar, tmpMaxSugar.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nAgentsInPlace, tmpNAgentsInPlace.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nextAgentIdx, tmpNextAgentIdx.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(pollution, tmpPollution.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(avePollution, tmpAvePollution.data(), size*size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(agentSugar, tmpAgentSugar.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(agentMetabolism, tmpAgentMetabolism.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(destinationIdx, tmpDestinationIdx.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(nextAgentIdx, tmpNextAgentIdx.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(destinationIdx, tmpDestinationIdx.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(agentSugar, tmpAgentSugar.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(agentMetabolism, tmpAgentMetabolism.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(pollution, tmpPollution.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(avePollution, tmpAvePollution.data(), size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpCurSugar.data(), curSugar, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpMaxSugar.data(), maxSugar, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpNAgentsInPlace.data(), nAgentsInPlace, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpNextAgentIdx.data(), nextAgentIdx, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpDestinationIdx.data(), destinationIdx, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpAgentSugar.data(), agentSugar, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpAgentMetabolism.data(), agentMetabolism, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpPollution.data(), pollution, size*size * sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(tmpAvePollution.data(), avePollution, size*size * sizeof(int), cudaMemcpyDeviceToHost);
copy(tmpNAgentsInPlace.begin(), tmpNAgentsInPlace.end(), ovNAgentsInPlace.begin()+bottom);
copy(tmpAgentSugar.begin(), tmpAgentSugar.end(), ovAgentSugar.begin()+bottom);
copy(tmpAgentMetabolism.begin(), tmpAgentMetabolism.end(), ovAgentMetabolism.begin()+bottom);
copy(tmpDestinationIdx.begin(), tmpDestinationIdx.end(), ovDestinationIdx.begin()+bottom);
copy(tmpNextAgentIdx.begin(), tmpNextAgentIdx.end(), ovNextAgentIdx.begin()+bottom);
copy(tmpPollution.begin(), tmpPollution.end(), ovPollution.begin()+bottom);
copy(tmpAvePollution.begin(), tmpAvePollution.end(), ovAvePollution.begin()+bottom);
prev = i;
}

if (false) {
    cout << "End of step sugar and ant distribution: " << endl;
    cout << "SUGAR LEVELS:" << endl;
    displayResult(ovCurSugar.data(), size, size*strands);
    cout << "ANT DISTRIBUTION:" << endl;
    displayResult(ovNAgentsInPlace.data(), size, size*strands);
    cout << "AGENT STORED SUGAR:" << endl;
    displayResult(ovAgentSugar.data(), size, size*strands);
}
}

if (false) {
    cout << "INITIAL SUGAR AND ANT DISTRIBUTION" << endl;
    cout << "SUGAR LEVELS:" << endl;
    displayResults(curSugar, size);
    cout << "ANT DISTRIBUTION:" << endl;
    displayResults(nAgentsInPlace, size);
    cout << "AGENT STORED SUGAR:" << endl;
    displayResults(agentSugar, size);
    cout << "AGENT METABOLISM:" << endl;
    displayResults(agentMetabolism, size);
}

// this synchronization call is not nesessary, if we delete
// it the synchronization will take place as part of cudaFree()
// call (time remains the same)

// Deallocate memory:
CATCH(cudaFree(curSugar));
CATCH(cudaFree(maxSugar));
CATCH(cudaFree(nAgentsInPlace));
CATCH(cudaFree(nextAgentIdx));
CATCH(cudaFree(destinationIdx));
CATCH(cudaFree(agentSugar));
CATCH(cudaFree(agentMetabolism));
CATCH(cudaFree(pollution));
CATCH(cudaFree(avePollution));

#include <chrono>
using namespace std::chrono;
int main(int argc, char const *argv[]) {
    cout << "Starting the GPU simulation" << endl;
    int size[] = { 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
    int max_time = 5;
    int i = 9;
    int strands = 5;

    auto start = high_resolution_clock::now();

    // for (int i=0; i< (sizeof(size)/sizeof(size[0])) ; i++) {
    int nAgents = size[i]*size[i]*5 / 5; // multiply by strands
    runDeviceSim(size[i], max_time, nAgents, strands);
    // }

    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "End of the GPU simulation" << duration.count() << endl;
    return 0;
}