1. **Introduction**

Between this quarter and the previous one, for my capstone project, I have been working on developing benchmarking programs for FLAME GPU2 to compare execution times and conduct a programmability analysis to compare with MASS CUDA. I was tasked with writing and analyzing the following benchmark programs:

* Game of Life
* Tuberculosis
* Neural Net

Between this quarter and the last, I was to port each of these programs from FLAME GPU to FLAME GPU2 and compare their metrics to those of the same benchmark programs written in MASS CUDA.

1. **Goals**

My goals for Fall and Winter quarters were as follows:

* Port Game of Life to FLAME GPU2
* Compare execution times for Game of Life between FLAME GPU2 and MASS CUDA
* Port Tuberculosis to FLAME GPU2
* Compare execution times for Tuberculosis between FLAME GPU2 and MASS CUDA
* Port Neural Net to FLAME GPU2

1. **Achievements**

The achievements over the Fall and Winter quarters were as follows:

* Understanding and modifying the example program, Game of Life provided with FLAME GPU2
* Running execution time tests and conducting a programmability analysis for Game of Life
* Porting Tuberculosis to FLAME GPU2
* Running execution time tests and conducting a programmability analysis for Tuberculosis
* Porting Neural Net to FLAME GPU2

Porting Neural Net came with substantial difficulties, as the original program took advantage of features that FLAME GPU2 did not have. This led to its development being extended and execution time tests could not be run in time. This is further expanded upon in the Neural Net section.

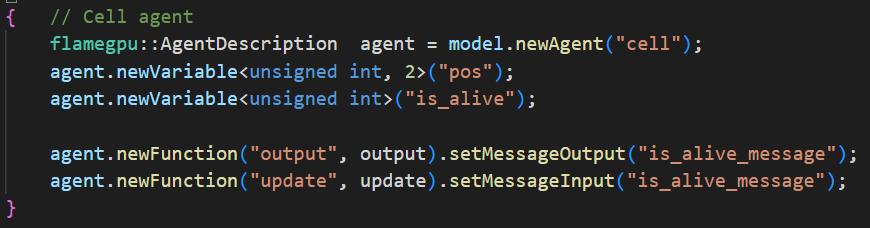
Due to the execution time tests for FLAME GPU2, MASS was able to be improved to be roughly equal in execution times.

* 1. **Implementation of Game of Life**

Game of Life, also known as Conway’s Game of Life, or just Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. Due to its nature of being a grid of cells interacting with their neighbors, it was not a surprise to find an implementation of it among the example programs provided with FLAME GPU2.

The code for Game of Life, and more generally a FLAME GPU2 program consisted of the following parts:

**Agent Specification**

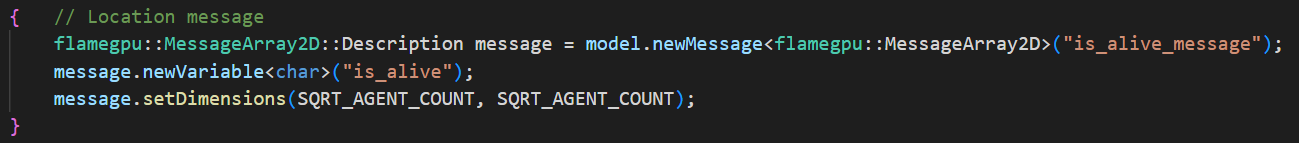


This code specifies a cell agent with a 2-element unsigned int array representing the agent’s position and an unsigned int representing if the agent is alive or dead.

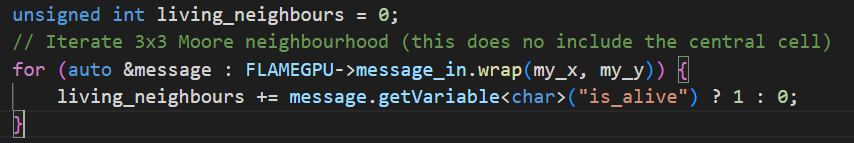
Two agent functions are then attached to the agent. An output function, where the cell communicates its state to its adjacent cells, and an update function, which takes in the communication from adjacent agents and decides the cell’s own new state.

The agent functions are attached to the agent while specifying what message that agent function will output and input, in this case the “is\_alive\_message.”

**Message Specification**

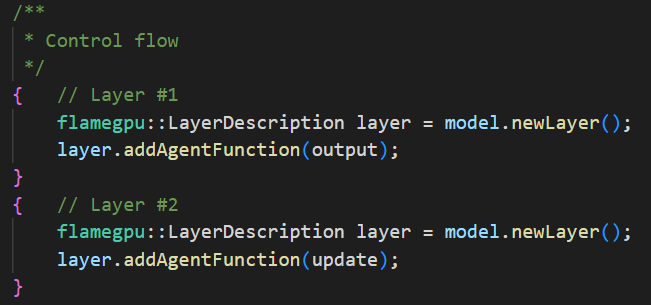


This code specifies the message sent by the “output” function and received by the “update” function. It is of the type MessageArray2D, which means that the message is a 2D array of messages where an agent function specifies that they are writing a message to a particular index within the 2D array. This allows the agent function receiving the message to look at a message at a particular location, or a set of messages relative to a particular location.



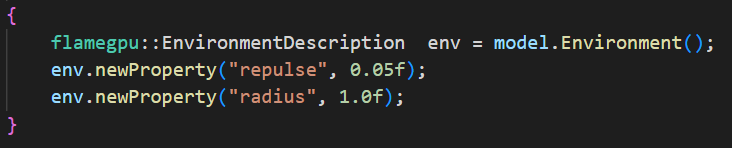
Here, in the update function, the wrap() method is used to iterate through messages from the cell’s Moore neighborhood centered on the cell’s own x and y coordinates.

**Layer Specification**

****

Here, control flow elements called layers are created, and agent functions are attached to them. Layers dictate the order in which agent functions are run, with a layer being created earlier being run earlier. In this case, two layers are created. The output function is attached to the first layer, and the update function is attached to the second. This means that the output function will execute first, sending each cell’s aliveness to its neighbors, followed by the update function executing, which will read in those messages and use them to determine each cell’s own state.

**Environment Specification**



Globally-accessible variables can be specified using the model’s environment. Agent functions can access these values at runtime.

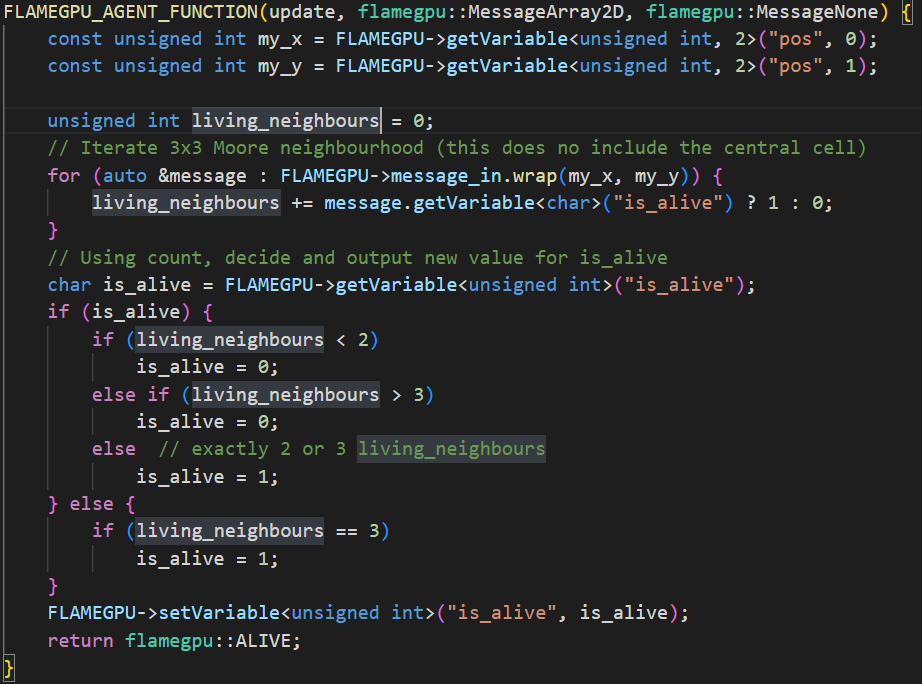
**Agent Functions**



Agent functions represent the behaviors of individual agents and contain the main part of the logic of a FLAME GPU2 program. The above code snippet is the output function of a cell in Game of Life. The function header specifies that it is an agent function, as distinct from the few other types of functions including host functions and init functions. The header also specifies the name of the function, the type of incoming message, which in this case is none, or MessageNone, and the type of outgoing message, which in this case is a MessageArray2D, as discussed earlier.

Within this agent function, FLAMEGPU->message\_out representing the outgoing message has two fields that are set. The variable “is\_alive”, as defined earlier, is set using the cell’s own “is\_alive” property. Then, as this is a MessageArray2D, the message’s index within the 2D array is set using the cell’s own position value.

This message is then passed on to the next agent function.



In the update function, the MessageArray2D is received and is iterated through using the wrap function, which accesses the location’s Moore neighborhood based on the cell’s own position, as shown earlier.

In order for an agent’s variables to be changed they must be set using FLAMEGPU->setVariable.

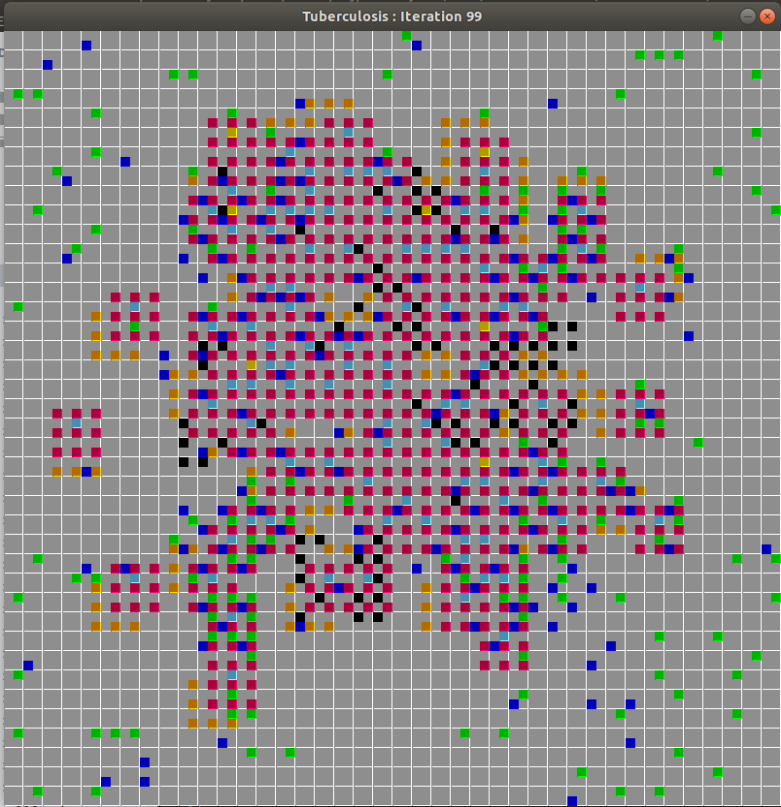
At the end of an agent function, flamegpu::ALIVE must be returned, representing that the agent is still alive at the end of the function execution. If the behavior is enabled when the agent function is initially assigned to the agent during agent specification, flamegpu::DEAD may be returned instead, indicating that the agent has died during the execution of the function.

* 1. **Implementation of Tuberculosis**

Tuberculosis is a simulation of Tuberculosis bacteria gradually infecting a human lung. The simulation consists of a 32 by 32 grid of spaces that represent the lung. 4 places in the center of the area are initially infected with bacteria, which gradually spreads to cover the lung.

Every simulation step, macrophage cells are spawned from predetermined blood vessels which wander the lung space somewhat randomly but will tend to move towards adjacent infected spaces if they detect a chemokine signal, which is generated by a place reacting to the presence of bacteria. Macrophages will consume bacteria on a space, making the space uninfected but infecting the macrophage in the process. Infected macrophages will gradually build up intracellular bacteria until they reach a particular threshold, at which point they are considered chronically infected. A chronically infected bacteria will eventually die when the intracellular bacteria reaches another, higher threshold. A macrophage that dies in this way will spread bacteria to nearby places.

After a certain number of simulation steps, by default 10, blood vessels will start to spawn T-cells, which will move in a similar manner to macrophages, attracted to a chemokine signal. When a macrophage and T-cell occupies the same space, one of two things will happen. If the macrophage is in its default state, or infected, the macrophage will become activated, and be able to consume bacteria from places without itself becoming infected again. If the macrophage is chronically infected when it encounters a T-cell, the macrophage will be killed.

This image is from the FLAME GPU (the original) implementation of Tuberculosis, as the visualizer for the FLAME GPU2 implementation is not yet complete. This is what the lung might look like after 99 simulation steps. 

The following sections will broadly explain the various agents and their specific behaviors.

**Place Agents**

Place agents represent a location on the lung. Place agents are responsible for managing the spread of bacteria, maintaining their chemokine signal, and if they are marked as a blood vessel, spawning in new macrophage and T-cell agents. Agent functions belonging to place agents are as follows:

* decay\_chemokine\_and\_grow\_bacteria
* cell\_recruitment
* approve\_macrophage\_movement
* approve\_tcell\_movement
* react\_to\_macro

**decay\_chemokine\_and\_grow\_bacteria**

This function is responsible for decrementing a place’s chemokine value at every simulation step, as well as indicating to adjacent places that they should grow bacteria if the day is a day for bacteria to grow, by default every tenth day.

**cell\_recruitment**

This function is responsible for creating new macrophage and T-cell agents, which is does depending on a number of factors including the current day, the presence or absence of a macrophage or T-cell currently on the place, and whether the day for T-cells to start spawning has passed.

**approve\_macrophage\_movement**

This function is responsible for responding to movement requests sent by macrophages. As only one macrophage can move to a place and two macrophages cannot occupy the same place, a place must approve the movement of a single macrophage agent to its own location.

**approve\_tcell\_movement**

This function is identical in functionality to approve\_macrophage\_movement, as T-cells share the same logic for moving as macrophages.

**react\_to\_macro**

This function is responsible for responding to the presence of macrophages and T-cells by adjusting the place agent’s bacteria values.

**Macrophage Agents**

Macrophage agents represent macrophage cells which roam the lung tissue and consume bacteria from places. Agent functions belonging to macrophage agents are as follows:

* macrophage\_request\_move
* macrophage\_move
* macrophage\_react

**macrophage\_request\_move**

This function is responsible for deciding which place a macrophage will move to. It does this by considering all the adjacent places and picks the one with the highest chemokine value. If all locations have a chemokine value of 0, the macrophage will move to a randomly selected place.

**macrophage\_move**

This function is responsible for receiving a message from the place agent’s approve\_macrophage\_movement. If a macrophage agent’s movement has been approved, this function will execute upon it, updating the macrophage’s position values.

**macrophage\_react**

This function is responsible for the macrophage reacting to the presence of bacteria and/or a T-cell at its new location, as well as updating intracellular bacteria if present. If the macrophage’s intracellular bacteria count is too high, or the macrophage is chronically infected and encounters a T-cell, this function will return flamegpu::DEAD, killing the agent.

**T-Cell Agents**

T-cell agents represent T-cells which move in the same manner as macrophages. T-cells are light on functionality as most of their interaction with other agents rely on other agents responding to their presence. Agent functions belonging to T-cell agents are as follows:

* tcell\_request\_move
* tcell\_move

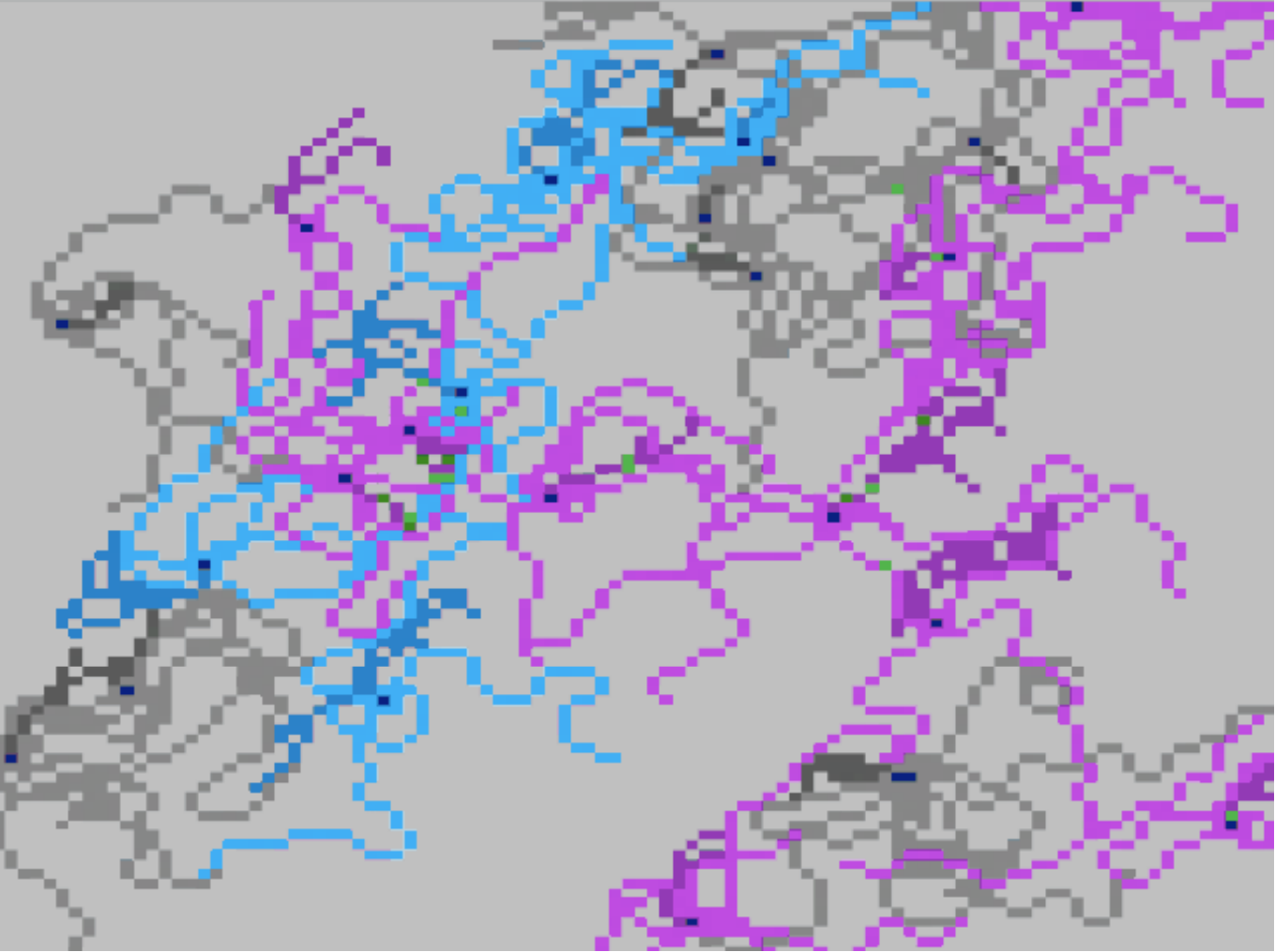
**tcell\_request\_move**

This function is identical in functionality to macrophage\_request\_move as they share the same movement behavior.

**tcell\_move**

This function is identical in functionality to macrophage\_move as they share the same movement behavior.

* 1. **Implementation of Neural Net**

Neural Net is a simulation of neurons growing and connecting to each other, eventually transmitting signals between them. The simulation space is a 100 by 100 grid of place agents that represent the portion of the brain which the program simulates. Upon initialization, each place has a 10% chance to become one of either a neutral, excitatory, or inhibitory neuron, or a 70% chance that the place will remain empty. If a neuron is determined to be created at a particular place, that place agent will be marked to contain a neuron, and a neuron agent will be created at that place’s coordinates.  


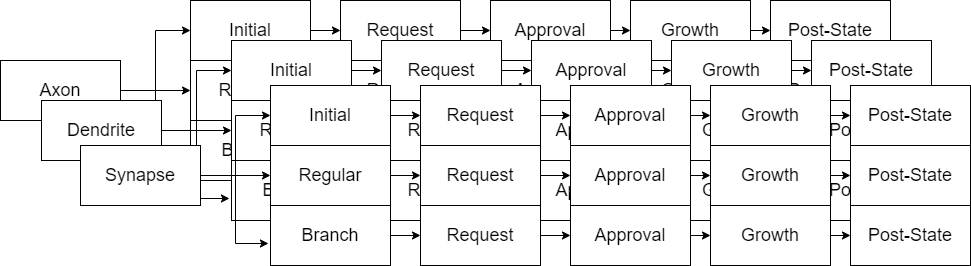
Each simulation step, each neuron will look through its growth order array, which was randomly shuffled and assigned when the neuron was created, and decide which direction to grow in. The growth order array contains eight elements, one of each of integers ranging from 0 to 7 in a random order. This array indicates the order the neuron will attempt to grow its auxiliary parts. The 0-index in the array is to be the direction the neuron will attempt to grow its axon, and the rest are directions where the neuron will attempt to grow its dendrites.

Once a neuron has decided to grow in a particular direction, it sends a growth request to the place agent at that location. The place agent then looks at all the growth requests it has received and sends a response to each of them. The place agent will approve a single request and deny the rest of them if it is unoccupied, or if it is occupied, it will deny all requests. If a neuron agent requesting to grow to a particular place receives approval from that place, it will create a new agent of the type it is growing at that location before notifying the place that a new agent now occupies that place.

This is where the complexities started to take hold. There are three ways in which a new neuron part agent can be created:

* When a neuron agent creates its first neuron part of its chain, e.g. when a neuron agent creates its initial axon part
* When a neuron part agent creates another one of itself, e.g. when a dendrite agent extends, creating another dendrite agent
* When a neuron part agent branches and creates a second one of itself, e.g. when an axon splits in two

Due to the limitations of FLAME GPU2, each of these neuron part relationships required their own agent functions. This means that for each of the neuron part types, and for each of the types of growth, four agent functions were required for each of the request, approval, growth, and post-state stages.

These limitations required that the neuron growth required, at minimum, 36 separate agent functions for each of the stages for each neuron part. Both designing this approach, implementing it and debugging it ate up significant development time, as whenever an issue was discovered in one of these agent functions, it had required that the same issue be fixed in each of the other 9 agent functions of that type.

Due to this, Neural Net unfortunately could not be completed as the specifications outlined in the time allotted. As a result, we decided to scale back the scope of the program by cutting the signal transmission between neurons.

Due to the repetitive nature of the agent functions, I will only go over the broad function of each of the categories of the functions within the growth chain. The agent types within Neural Net are as follows:

**Place Agents**

Place agents represent a location on the brain. Place agents are responsible for tracking what type of neuron part is present at that location, as well as using that information to respond to growth requests. Agent functions belonging to place agents are as follows:

* approve\_growth
* post\_state

**approve\_growth**

This category of function receives growth requests from the various neuron parts. If the place is already occupied by a neuron part, the function will deny all requests. If the place is unoccupied, the place will approve the first request and deny the rest.

**post\_state**

This category of function receives a message from the growth function and updates the place’s variables to indicate that it now is occupied by a neuron part. This information is used in future simulation steps when approving or denying neuron growth.

**Neuron Agents**

Neuron agents represent the center of a neuron. The neuron agent is responsible for tracking the order for which it will grow the initial part in the chain of each of the surrounding neuron parts, that is, the axon and dendrites. Agent functions belonging to place agents are as follows:

**make\_growth\_decisions**

This agent function consults the growth order array and decides which neuron part to grow during this simulation step.

**request\_to\_grow\_initial\_[neuron part]**

This category of agent function makes a request to a place agent to grow the first axon or dendrite agent attached to the neuron.

**grow\_initial\_[neuron part]**

This category of agent function receives the growth request response from the place agent and grows the first axon or dendrite agent attached to the neuron.

**Axon, Dendrite, and Synapse Agents**

These agents represent an individual part of the neuron. They form chains of themselves starting at the neuron agent. At each growth step, they have a random chance to cease their growth. When an axon ceases its growth it will instead attempt to grow a synapse agent. Agent functions shared between these agents are as follows:

**make\_[neuron part]\_growth\_decisions**

This category of agent function decides if this agent will cease to grow, if it will branch, and calculate the position of the place it will attempt to grow and/or branch to.

**request\_to\_grow\_[neuron part]**

This category of agent function makes a request to a place agent to grow another of the corresponding part type.

**grow\_[neuron part]**

This category of agent function receives the growth request response from the place agent and grows the corresponding part type.

**request\_to\_grow\_[neuron part]\_branch**

This category of agent function makes a request to a place agent to grow a branch of the corresponding part type.

**grow\_[neuron part]\_branch**

This category of agent function receives the growth request response from the place agent and grows the branch of the corresponding part type.

1. **Evaluation**
   1. **Game of Life**

**Game of Life Execution Times**

The following runtime data was collected by running Game of Life in both MASS CUDA and FLAME GPU2 for three game sizes of 165 by 165 places, 1000 by 1000 places, and 2000 by 2000 places for 250 steps.

**MASS CUDA**

| 165 \* 165 | 1000 \* 1000 | 2000 \* 2000 |
| --- | --- | --- |
| 3,349 ms | 4,441 ms | 14,688 ms |

**FLAME GPU2**

| 165 \* 165 | 1000 \* 1000 | 2000 \* 2000 |
| --- | --- | --- |
| 31 ms | 113 ms | 359 ms |

The results indicated that for all sizes, FLAME GPU2 was around two orders of magnitude faster than MASS CUDA. This data led to MASS being improved. The improved execution times are as follows.

**MASS CUDA V2**

| 500 \* 500 | 1000 \* 1000 | 2000 \* 2000 |
| --- | --- | --- |
| 47 ms | 135 ms | 481 ms |

**Game of Life Programmability Analysis**

The following programmability metrics were collected by using Lizard, a Cyclomatic Complexity Analyzer for many languages including C++. In order for Lizard to accept the MASS CUDA and FLAME GPU2 programs, they had to be renamed to be .cpp files instead of .cu, which did not appear to affect Lizard’s functionality.

**MASS CUDA**

| LOC | Cyclomatic Complexity | Boilerplate code | Boilerplate % |
| --- | --- | --- | --- |
| 147 | 2.6 | 8 | 5.4% |

**FLAME GPU2**

| LOC | Cyclomatic Complexity | Boilerplate code | Boilerplate % |
| --- | --- | --- | --- |
| 114 | 4 | 78 | 68% |

The results indicated that FLAME GPU2 had an overwhelmingly higher amount of boilerplate code, which was consistent with my experiences in looking over Game of Life and writing Tuberculosis, where comparatively little of the code specified agent behavior. This was largely because in a FLAME GPU2 program, most elements such as agents, agent functions, and messages between agent functions have to have all of their properties specified beforehand, leading to many lines of code specifying their functionality before their actual use.

* 1. **Tuberculosis**

**Tuberculosis Programmability Analysis**

**FLAME GPU2**

| LOC | Cyclomatic Complexity | Boilerplate code | Boilerplate % |
| --- | --- | --- | --- |
| 1057 | 4.97 | 265 | 25% |

Tuberculosis was a much longer and more intricate program than Game of Life, as indicated by its total lines of code. Each of the agents had much more complex behaviors, as denoted by the higher cyclomatic complexity.

Interestingly, the boilerplate code, while roughly four times greater than Game of Life, made up a smaller portion of the program. This is because, while like in Game of Life, the agent, message, and control flow specification took up a large amount of space, unlike Game of Life, the agent behaviors were, again, much more complex, taking up comparatively more lines of code.

It should be noted that the original implementation of Tuberculosis for FLAME GPU had taken advantage of many features present in FLAME GPU that were not in FLAME GPU2. Primarily, this was the functionality to receive multiple types of message originating from multiple different agent functions in a single agent function. To recreate the behaviors of the original program in FLAME GPU2, I had to add multiple extra agent functions to send and receive information at intermediate steps between the ported functions and save that data as properties within the agent. This increased the complexity of the FLAME GPU2 implementation by a significant amount.

1. **Conclusion**

To summarize, over the last two quarters, I have worked on developing the following benchmarking programs for FLAME GPU2:

* Game of Life
* Tuberculosis
* Neural Net

Game of Life and Tuberculosis have both been completed, and execution time and programmability metrics for them have been collected. Unfortunately, Neural Net was not able to be completed in the allotted time frame due to the aforementioned issues, so I will be working on completing it next quarter.

**References:**

* Gardner, Martin (October 1970). *"The fantastic combinations of John Conway's new solitaire game 'life'" (PDF). Mathematical Games. Scientific American.* Vol. 223, no. 4. pp. 120–123. doi:10.1038/scientificamerican1070-120. JSTOR 24927642. Archived (PDF) from the original on 2022-10-09.
* D’Souza, R. et al(n.d.). *Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis On Graphics Processing Units.* Michigan Tech. University*.*
* Brain Grid: Self-Organizing Neural Network, “Accessed on: January 1, 2024. [Online]. Available: <https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/dev_base/Specifications/SelfOrganizingNet.docx>.”
* FLAME GPU 2 Userguide and API Documentation, “Accessed on: October 6, 2023. [Online]. Available: <https://docs.flamegpu.com/>.”

**Appendix A: Code**

The implementations for Game of Life and Tuberculosis can be found at the following locations:

| Game of Life | https://github.com/FLAMEGPU/FLAMEGPU2/tree/master/examples/cpp/game\_of\_life |
| --- | --- |
| Tuberculosis | https://bitbucket.org/alexhilwa/flamegpu2\_tuberculosis/src/main/ |
| Neural Net | https://bitbucket.org/alexhilwa/flamegpu2\_neuralnet/src/main/ |

**Appendix B: How to Run**

In order to run Game of Life, follow these instructions:

1. Download the FLAME GPU2 library from:
   1. <https://github.com/FLAMEGPU/FLAMEGPU2/tree/master>
2. Unzip FLAMEGPU2-master
3. Mkdir -p build && cd build
4. cmake .. -DCMAKE\_CUDA\_ARCHITECTURES=61 -DCMAKE\_BUILD\_TYPE=Release
5. cmake --build . --target game\_of\_life -j 8
6. ./bin/Release/game\_of\_life --verbose --steps 10
   1. This command will run the program in verbose mode, for 10 steps.
   2. Using the --help command will explain argument usage.

In order to run Tuberculosis, follow these instructions:

1. Download the FLAME GPU2 library from:
   1. <https://github.com/FLAMEGPU/FLAMEGPU2/tree/master>
2. Unzip FLAMEGPU2-master
3. In FLAMEGPU2-master/examples/cpp add the **tuberculosis** directory from the BitBucket link
4. Return to FLAMEGPU2-master
5. Mkdir -p build && cd build
6. Edit FLAMEGPU2-master/CMakeLists.txt
   1. Under the section: **# Options to enable building individual examples, if FLAMEGPU\_BUILD\_ALL\_EXAMPLES is off.**
   2. Include the line: **cmake\_dependent\_option(FLAMEGPU\_BUILD\_EXAMPLE\_TUBERCULOSIS "Enable building examples/cpp/tuberculosis" OFF "FLAMEGPU\_PROJECT\_IS\_TOP\_LEVEL; NOT FLAMEGPU\_BUILD\_ALL\_EXAMPLES" OFF)**
   3. Under the section: **# Add each example**
   4. Include the lines:
   5. **if(FLAMEGPU\_BUILD\_ALL\_EXAMPLES OR FLAMEGPU\_BUILD\_EXAMPLE\_TUBERCULOSIS)**
   6. **add\_subdirectory(examples/cpp/tuberculosis)**
   7. **endif()**
   8. Alternatively, find an already edited version of the CMakeLists.txt in the root directory of the Tuberculosis BitBucket
7. cmake .. -DCMAKE\_CUDA\_ARCHITECTURES=61 -DCMAKE\_BUILD\_TYPE=Release
8. cmake --build . --target tuberculosis -j 8
9. ./bin/Release/tuberculosis --verbose --steps 100

In order to run Tuberculosis, follow these instructions:

1. Download the FLAME GPU2 library from:
   1. <https://github.com/FLAMEGPU/FLAMEGPU2/tree/master>
2. Unzip FLAMEGPU2-master
3. In FLAMEGPU2-master/examples/cpp add the **neural\_net** directory from the BitBucket link
4. Return to FLAMEGPU2-master
5. Mkdir -p build && cd build
6. Edit FLAMEGPU2-master/CMakeLists.txt
   1. Under the section: **# Options to enable building individual examples, if FLAMEGPU\_BUILD\_ALL\_EXAMPLES is off.**
   2. Include the line: **cmake\_dependent\_option(FLAMEGPU\_BUILD\_EXAMPLE\_NEURAL\_NET "Enable building examples/cpp/neural\_net" OFF "FLAMEGPU\_PROJECT\_IS\_TOP\_LEVEL; NOT FLAMEGPU\_BUILD\_ALL\_EXAMPLES" OFF)**
   3. Under the section: **# Add each example**
   4. Include the lines:
   5. **if(FLAMEGPU\_BUILD\_ALL\_EXAMPLES OR FLAMEGPU\_BUILD\_EXAMPLE\_NEURAL\_NET**
   6. **add\_subdirectory(examples/cpp/neural\_net)**
   7. **endif()**
   8. Alternatively, find an already edited version of the CMakeLists.txt in the root directory of the Neural Net BitBucket
7. cmake .. -DCMAKE\_CUDA\_ARCHITECTURES=61 -DCMAKE\_BUILD\_TYPE=Release
8. cmake --build . --target neural\_net -j 8
9. ./bin/Release/neural\_net --verbose --steps 100