Alex Hilwa
CSS 497 A Term Report Fall 2023

1. Introduction

This quarter, for the first five credits of my capstone project, I am 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 next, I am 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.

2. Goals

My goals for Fall quarter 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

Of the benchmarking programs, Tuberculosis was by far the most complex and was expected to take until partway through Winter quarter, next year.

3. Achievements

This quarter's achievements were as follows:

- Understanding how to install and run FLAME GPU2
- Learning the basics of programming in FLAME GPU2
- Understanding the code in the example programs and reading documentation
- Running execution time tests and conducting a programmability analysis for Game of Life
- Understanding the FLAME GPU implementation of Tuberculosis
- Implementing Tuberculosis in FLAME GPU2

Within FLAME GPU2's code was provided a set of example programs that demonstrated how to write and run code for it. Among these example programs was an implementation of Game of Life. Because of this, I was able to run to execution time comparisons and programmability analysis without porting the existing FLAME GPU program. This allowed for me to start on porting Tuberculosis early, and I was able to finish most of the work before the end of the quarter.

4. Results

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.

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.

Tuberculosis Programmability Analysis

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.

5. Next Quarter's Plan

With this quarter's goals having been progressed through ahead of schedule, next quarter's goals have also been moved up. Previously, I was aiming to complete Tuberculosis early into next quarter along with a visualizer, but now the new goals are as follows:

- Complete Tuberculosis visualizer
- Understand FLAME GPU implementation of Neural Net
- Port Neural Net to FLAME GPU2
- Collect execution time data and conduct programmability analysis on FLAME GPU2 implementation of Neural Net

For the Tuberculosis visualizer, the FLAME GPU2 program outputs a series of log files which describe the state of the simulate at each step. At its current state, the visualizer is able to read in and interpret the log files. All that is left to do is to use the information from the logs to construct a visual representation of the Tuberculosis simulate that can be stepped through.

6. Implementation: 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

```
{   // Cell agent
    flamegpu::AgentDescription  agent = model.newAgent("cell");
    agent.newVariable<unsigned int, 2>("pos");
    agent.newVariable<unsigned int>("is_alive");

    agent.newFunction("output", output).setMessageOutput("is_alive_message");
    agent.newFunction("update", update).setMessageInput("is_alive_message");
}
```

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

```
{   // Location message
    flamegpu::MessageArray2D::Description message = model.newMessage<flamegpu::MessageArray2D>("is_alive_message");
    message.newVariable<char>("is_alive");
    message.setDimensions(SQRT_AGENT_COUNT, SQRT_AGENT_COUNT);
}
```

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.

```
unsigned int living_neighbours = 0;
// Iterate 3x3 Moore neighbourhood (this does no include the central cell)
for (auto &message : FLAMEGPU->message_in.wrap(my_x, my_y)) {
    living_neighbours += message.getVariable<char>("is_alive") ? 1 : 0;
}
```

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

```
/**
 * Control flow
 */
{   // Layer #1
    flamegpu::LayerDescription layer = model.newLayer();
    layer.addAgentFunction(output);
}
{   // Layer #2
    flamegpu::LayerDescription layer = model.newLayer();
    layer.addAgentFunction(update);
}
```

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

```
{
    flamegpu::EnvironmentDescription  env = model.Environment();
    env.newProperty("repulse", 0.05f);
    env.newProperty("radius", 1.0f);
}
```

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

Agent Functions

```
FLAMEGPU_AGENT_FUNCTION(output, flamegpu::MessageNone, flamegpu::MessageArray2D) {
    FLAMEGPU->message_out.setVariable<char>("is_alive", FLAMEGPU->getVariable<unsigned int>("is_alive"));
    FLAMEGPU->message_out.setIndex(FLAMEGPU->getVariable<unsigned int, 2>("pos", 0), FLAMEGPU->getVariable<unsigned int, 2>("pos", 1));
    return flamegpu::ALIVE;
}
```

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.

```
FLAMEGPU_AGENT_FUNCTION(update, flamegpu::MessageArray2D, flamegpu::MessageNone) {
    const unsigned int my_x = FLAMEGPU->getVariable<unsigned int, 2>("pos", 0);
    const unsigned int my_y = FLAMEGPU->getVariable<unsigned int, 2>("pos", 1);

    unsigned int living_neighbours = 0;
    // Iterate 3x3 Moore neighbourhood (this does no include the central cell)
    for (auto &message : FLAMEGPU->message_in.wrap(my_x, my_y)) {
        living_neighbours += message.getVariable<char>("is_alive") ? 1 : 0;
    }
    // Using count, decide and output new value for is_alive
    char is_alive = FLAMEGPU->getVariable<unsigned int>("is_alive");
    if (is_alive) {
        if (living_neighbours < 2)
            is_alive = 0;
        else if (living_neighbours > 3)
            is_alive = 0;
        else  // exactly 2 or 3 living_neighbours
            is_alive = 1;
    } else {
        if (living_neighbours == 3)
            is_alive = 1;
    }
    FLAMEGPU->setVariable<unsigned int>("is_alive", is_alive);
    return flamegpu::ALIVE;
}
```

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.

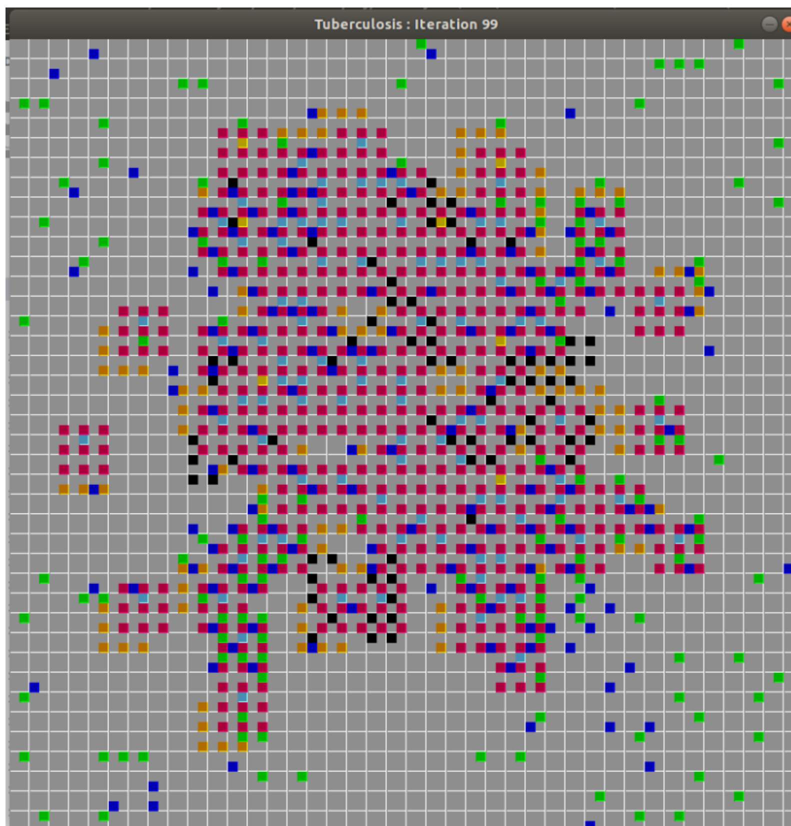
Alex Hilwa
CSS 497 A Term Report Fall 2023

7. Implementation: 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.

Alex Hilwa
CSS 497 A Term Report Fall 2023

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/

Appendix B: How to Run

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

1. Download the FLAME GPU2 library from:
    a. 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
    a. This command will run the program in verbose mode, for 10 steps.
    b. Using the --help command will explain argument usage.

In order to run Tuberculosis, follow these instructions:

1. Download the FLAME GPU2 library from:
    a. 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
    a. Under the section: **# Options to enable building individual examples, if FLAMEGPU_BUILD_ALL_EXAMPLES is off.**
    b. 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)**
    c. Under the section: **# Add each example**
    d. Include the lines:
    e. if(FLAMEGPU_BUILD_ALL_EXAMPLES OR FLAMEGPU_BUILD_EXAMPLE_TUBERCULOSIS)
    f. add_subdirectory(examples/cpp/tuberculosis)
    g. endif()

h. 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