Development of MASS CUDA and

FLAME GPU2 Benchmarks

Paul Adrian Sese

CSS 497: Capstone

2024 Winter Quarter Term Report

3/19/2025

Contents

[1. Introduction 3](#_Toc193806593)

[1.1 Motivation 3](#_Toc193806594)

[1.2 Project Goal 3](#_Toc193806595)

[2. Background 4](#_Toc193806596)

[2.1 Virtual Development Team (VDT) 4](#_Toc193806597)

[2.2 Challenges 5](#_Toc193806598)

[3. Progress 7](#_Toc193806599)

[3.1 2024 Fall Quarter 7](#_Toc193806600)

[3.2 2024 Winter Quarter 7](#_Toc193806601)

[3.3 MASS CUDA VDT Implementation 7](#_Toc193806602)

[3.4 FLAME GPU 2 Implementation 8](#_Toc193806603)

[4. Code Snippets 10](#_Toc193806604)

[4.1 MASS CUDA VDT 10](#_Toc193806605)

[4.2 FLAME GPU 2 11](#_Toc193806606)

[5. Evaluation 14](#_Toc193806607)

[5.1 Performance 14](#_Toc193806608)

[5.2 Programmability 16](#_Toc193806609)

[6. Conclusion 17](#_Toc193806610)

[6.1 Next Steps 17](#_Toc193806611)

[7. Appendix 18](#_Toc193806612)

[7.1 MASS CUDA 18](#_Toc193806613)

[7.2 FLAME GPU 2 19](#_Toc193806614)

# 1. Introduction

## Motivation

With MASS CUDA still in development, these benchmarks are beneficial to illustrate MASS CUDA’s strengths and weaknesses. They highlight essential features that should be maintained in future additions as well as capabilities that should be added down the line. FLAME GPU 2 is a similar library to MASS CUDA and comparing both libraries’ performances and coding processes against each other can help determine what direction to improve MASS CUDA in. Finding what features make FLAME GPU 2 perform better than MASS CUDA can help current and future developers learn about possible additional features to take from FLAME GPU 2 or even adapt or improve current features to include their benefits.

## Project Goal

For my capstone project, I was assigned with creating VDT simulations for both MASS CUDA and FLAME GPU 2. My goal for the 2024 winter quarter was to add more functionality options to MASS CUDA’s version to ensure it runs similarly to FLAME GPU 2’s version. Another goal was to implement VDT for FLAME GPU 2 and to add timers to both versions and compare them both in performance and programmability.

# 2. Background

## 2.1 Virtual Development Team (VDT)

The simulation and its description have not changed since last quarter as there was no need to make any changes.

This simulation revolves around intra-team communication. It is made up of software engineering agents which make up teams that complete tasks that they are assigned. Each team is made up of 25 engineer agents and their tasks. There are 5 types of engineers, and their type determines when they will work on each task. The types of engineers are:

1. Project Lead
2. UX Designer
3. Senior Software Developer
4. Junior Software Developer
5. Test Engineer

Teams can have different configurations of the amount of each type of engineer they have(Ex. One configuration could be 5 of each type of engineer and another could be 21 project leads and one of the other types of engineers).

There are two types of tasks, production tasks and collaboration tasks. Production tasks are worked on by one engineer at a time and collaboration tasks are worked on by multiple engineers at a time. Production tasks go through the hierarchy of engineers in the order listed above. It is first worked on by the project lead, followed by the UX designer, followed by the other types of developers until the test engineer completes the task. Collaboration tasks on the other hand are worked on by multiple engineers simultaneously. Each one requires at least one type of engineer required to work together and complete the task together. Engineers currently working on a collaboration task cannot work on another production task until the collaboration task is completed. Collaboration tasks have one more detail compared to production tasks being their start day as collaboration tasks can’t be worked on by a team until after or on a certain day.

Engineers receive a production task as soon as they both don’t have one currently assigned to them and there is one available for them to work on requiring the engineer’s type’s expertise. They are randomly assigned from the available tasks in the following manner:

* 50% based on priority
* 20% based on oldest (first task placed in tray)
* 10% randomly
* 20% most recent item added to available tray

When an engineer receives a task, it hasn’t yet received before it must process the task. When processing a task, there is a 30% chance of an exception where if they aren’t a lead they return the task with more hours required from an engineer one level higher in the hierarchy. If they are a lead once processed, they need to work on the task twice as long as they originally had to before returning it.

The simulation ends once all teams complete all their tasks and the time it took for each team to complete their tasks is displayed.

## 2.2 Challenges

One of the main purposes of the simulation is to compare the two libraries’ performances running them. To get the most accurate results the data and the processes between both versions need to be consistent. This makes features that involve RNG in the simulation to create possible outliers in execution times. These features include deciding which task to assign affecting the order of tasks that are completed as well as processing tasks possibly adding more hours needed to finish the task. These functions must be ignored or skipped in order to have controlled and consistent simulations. This wasn’t an issue for FLAME GPU 2 since there wasn’t a good object-oriented way to add them.

 Another challenge was like a challenge faced developing MASS CUDA’s version discussed in the last term paper with CUDA not having access to the use of Standard Template Libraries (STL). This becomes an issue when having to store tasks as well as take them out and place them in trays throughout the simulation as tasks move down the hierarchy of trays. This brings up another challenge with FLAME GPU 2 specifically for this benchmark being that the library only allows the storage of variables that are only primitive types such as int, float, etc. This limits developers preventing them from defining their own storage types and other objects including tasks which are a major part of this simulation. FLAME GPU 2’s agents primitive variables can only be at most 1D arrays. This compounds with the other challenges, especially with how much data each task stores and is needed to be accessed and modified throughout the simulation.

# 3. Progress

## 3.1 2024 Fall Quarter

After this quarter MASS CUDA’s version of VDT has been implemented and tested. The program is limited by how many tasks in total it can handle with both the GPU’s limited storage and the task trays’ implementation almost always being allocated more space than it needs to handle all different types of tasks and team configurations. MASS CUDA’s files have been documented and commented on along with a README file instructing future developers on how to run the simulation.

## 3.2 2024 Winter Quarter

After this quarter MASS CUDA’s version was modified to have more options running the simulation, adding timers with the option to print execution results. Results printed are overall execution time, time per hour(step in the simulation), and time for each function in the step. The first two are important for comparing FLAME GPU 2 with the latter to be used for comparison of a later version MASS CUDA. FLAME GPU 2 was developed and tested with both versions having similar outputs when running with the same inputs. Both versions were tested with performance logged as well as programmability statistics calculated with cyclomatic complexity determined using Lizard.

## 3.3 MASS CUDA VDT Implementation

No major changes have been made to MASS CUDA’s version of VDT since last quarter. RNG variable was added for teams to determine if the simulation will run with features that involve RNG such as teams determining which task in the tray to assign as well as engineers processing their tasks with the possibility of adding more hours to the task. RNG variable is determined by input running the simulation and is off by default ignoring features that involve RNG. Timers have been added and like the RNG option is determined as an argument running the simulation and by default is off. When on performance execution times are printed at the end of the simulation.

## 3.4 FLAME GPU 2 Implementation

FLAME GPU 2’s version of VDT is very similar to MASS CUDA’S implementation. FLAME GPU 2 only has agents so both engineers and teams are agents as in MASS CUDA places are just a different type of agent in a way. The simulation takes its input from a json file which is created in the Input folder which reads files that MASS CUDA’s versions read including the task file as well as the team configuration file. The input file also takes in numTeams which is how many teams of each configuration will be generated for the simulation.

Because FLAME GPU 2 only stores primitive types and doesn’t allow user defined classes tasks in this simulation are represented as arrays. The first 5 elements in the array store are the hour for the 5 levels of engineers with collaboration tasks having a 6th element representing the day the task can be started. Due to how limited the library is in terms of user-defined classes the simulation’s design is simpler compared to MASS CUDA’S.

One example is how tasks are stored. In MASS CUDA each team gets a copy of their tasks. While in FLAME GPU 2’s version the tasks are stored in the environment which are global variables in the simulation and can only be accessed by agents such as the teams. They are stored as macro properties which allow the use of 2D arrays with the first index pointing to the task and the second index pointing to the element of the corresponding task.

Since the tasks and their data are stored in the environment teams don’t store all the task data but their indices pointing to the macro property element. These indices are stored in the teams’ trays which are implemented as deques storing the indices in arrays of integers and keeping track of the trays with other integer arrays including the front, back, and the number of tasks in the tray. To stay consistent with MASS CUDA’s VDT without RNG features, this version has teams assign tasks in front of the tray and engineers don’t have to process tasks. This is also implemented this way because of how tasks are stored, not keeping much track of hours for each task and how processing tasks can add more hours.

In MASS CUDA engineers and teams communicate and assign/submit tasks by having teams simply accessing the engineers that reside in their place. FLAME GPU 2 on the other hand handles agent communication via messages. This simulation specifically uses array and 2D array messaging. Engineers write their data in 2D array messages with the first dimension corresponding to the index of the team they’re in and the second dimension corresponding to their index within their team. Teams read from these arrays and write to an array message with its index corresponding to the team’s index. These messages’ variables are arrays which like 2D arrays that have the engineers’ indices point to which element in the array to write/read.

There were issues with printing the how long each team took to finish all the tasks, so teams and their times are printed once they finish. The team agents are killed after they print their results, and their corresponding engineers are killed after not receiving any data in their messages. The simulation finishes once all team agents have been killed.

# 4. Code Snippets

## 4.1 MASS CUDA VDT



Figure 1:Team Attributes



Figure 2: Initialization Timer



Figure 3: Hour(Step) Timer



Figure 4: Overall Timer

## 4.2 FLAME GPU 2



Figure 5:Environment Properties



Figure 6:Team Variables



Figure 7:Team Functions



Figure 8:Engineer Variables



Figure 9:Engineer Functions

# 5. Evaluation

Performance is simpler to analyze with MASS CUDA significantly outperforming FLAME GPU 2 even with the latter having a simpler design. MASS CUDA having the better initialization time was expected but average hour and overall time was a surprise. FLAME GPU 2’s average execution time per hour was significantly greater than MASS CUDA. This is most likely due to FLAME GPU 2’s exit condition requiring the program to download all team data to the main memory to check if there are any teams that aren’t finished after each hour. While MASS CUDA simply downloads one attribute from each team instead of all their data per hour. The average hour execution time causes the greater overall execution time for FLAME GPU 2.

Programmability on the other hand doesn’t have a clear winner. MASS CUDA being the more complex version of the program having almost 4 times as many lines of code as FLAME GPU 2. This isn’t that surprising as MASS CUDA allows the definition of tasks, deques for trays, and a set for processed tasks. Even with all those LOC MASS CUDA has a lower cyclomatic complexity illustrating that MASS CUDA is simpler in terms of code. MASS CUDA also a higher percentage of boilerplate code which makes it easier to program compared to FLAME. FLAME on the other hand has a lower Lack of Cohesion of Method(LCOM) which makes modifying FLAME’s code less complex considering having to change fewer methods if a variable is affected by change in code or design.

## Performance

This data was recorded with 100 tasks and four different team configurations.

Figure 10:Initialization

Figure 11:Average Hour(Step)

Figure 12:Overall

## Programmability

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Framework | LoC | Cyclomatic Complexity | Bolerplate LoC | Boilerplate % | LCOM |
| MASS CUDA | 1268 | 2.9 | 227 | 17.9022082 | 0.676551 |
| FLAME GPU 2 | 376 | 7.1 | 17 | 4.521276596 | 0.478261 |

# Conclusion

In the last two quarters I was able to develop Virtual Development Team (VDT) simulation programs for both GPU libraries MASS CUDA and FLAME GPU 2. MASS CUDA was learned primarily focused on past white papers and comments on library code. FLAME GPU 2 has various examples in their documentation which were primarily used. Other benchmark programs were helpful for both libraries. Both benchmarks were analyzed for performance and programmability using timers, Lizard to analyze lines of code and complexity, along with manually calculating other factors in programmability. MASS CUDA’s features closely align with MASS cpp’s version while FLAME GPU 2’s is simpler to test the libraries with FLAME GPU 2’s limitations and restrictions. Both versions have the same output and process making analysis results accurate for comparing libraries.

## 6.1 Next Steps

MASS CUDA has various benchmarks that are incomplete or outdated. Warren Liu’s version adds the use of set and get attributes and ignoring place states made a significant difference in terms of code. These outdated programs require a lot of rewriting. More specifically, MASS CUDA’s Bail in Bail Out hasn’t been completed in an older version and FLAME GPU 2’s version is yet to be developed.

In terms of MASS CUDA library one major thing that should be maintained in future versions is the ability for agents to store user-defined classes, this features give it a significant advantage compared to FLAME GPU 2 in terms of programmability. The ability to download only one attribute from agents is also a nice feature to maintain.

# Appendix

## 7.1 MASS CUDA

Explanation for each file in the project:

**src:**

VDT.cu & VDT.h: Main file of the program reading the input and creating/managing the agents and places.

Team.cu & Team.h: Place file defining the team that store the tasks and assign them to their engineers.

Engineer.h & Engineer.cu: Agent file defining an engineer that adds itself to its team and works on tasks

Tash.h & Task.cu: Task file representing tasks to be completed storing how much time it’s required to be completed.

Deque.h: Double ended queue implementation using a circular array for task trays.

Set.h: Set implementation using linked nodes to store the integer task IDs.

main.cu: Main that runs the program

Makefile: Used to download the necessary MASS files and compile code to ccreate the executable to run.

To run the program on juno, follow the following steps from the project directory(Similar to information on readme):

**Download MASS files**

make develop

**Make the executable**

make build

**Run the program**

./bin/VDT

## 7.2 FLAME GPU 2

Explanation for each file in the project:

**src:**

main.cu: Main that runs the program

**Input:**

generateInput: Generate a json file taking arguments(numteams, taskFile, configFile)

Makefile: Used to download the necessary MASS files and compile code to ccreate the executable to run.

To run the program on juno, follow the following steps from the project directory(Similar to information on readme):

**Create the build directory and change into it**

mkdir -p build && cd build

**Configure CMake**

cmake .. -DCMAKE\_BUILD\_TYPE=Release-DCMAKE\_CUDA\_ARCHITECTURES=86

**Build the targets**

cmake –build . –target flamegpu VDT -j 8

**Run the program**

./bin/Release/VDT