Programming Abstractions for GPU-Accelerated Agent-Based Simulations

by

Bailey Hayward Sostek

A Thesis Proposal

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

February 2023

1 Introduction

Using the GPU optimally to accelerate agent-based simulations is an area which currently requires specialized programming knowledge. For example, in FlameGPU[10] a modern simulation framework, a user is required to write code in Python, C++, and CUDA to create GPU accelerated simulations. We believe that it is instead possible to create a single graphical high-level language to author such simulations. This would alleviate the need for simulation authors to have a deep understanding of modern graphics APIs or for them to need to program in multiple languages concurrently. The research questions which we want to answer in this thesis are:

1. What are the comparative computational advantages and disadvantages of using a high-level language to create GPU-accelerated simulations?
2. What are the limitations of using a high-level language to create and describe agent based simulations when compared to low-level languages? Which kinds of simulations would not be able to be represented with a high-level language which would be able to be represented in a low-level language?

We begin addressing these questions with evaluations of the performance and limitations of existing agent-based simulation frameworks. We then present investigations into modern graphics frameworks and APIs exploring how their modern features could be used to accelerate agent-based simulations through parallelizing workloads. We end by describing the design of a node-based programming environment for GPU-accelerated simulations that we are creating, named *Apiary*, and provide a timeline for the work that remains to be completed.

2 Background

In this section we describe the strengths and weaknesses of current popular agent-based simulation frameworks. This research informed which features our simulation engine would need to implement in order to be comparable to modern tools.

The characteristics of the simulation frameworks we investigated were the following:

1. Is the simulation framework is using GPU acceleration?

2. Are there limitations as to the maximum number of Agents the framework can represent at one time and if so how is this information communicated with the user?

3. Is the user required to write out their simulations with code? If so, how many languages do they needed to write in concurrently to describe and create their simulation?

With these questions in mind, we investigated a variety of simulation frameworks to explore how they intended a user to interact with their frameworks. The simulation frameworks we chose to investigate were all described as tools which enable the development of agent based simulations for different areas of industry and research.

2.1 Simulation Tools

2.1.1 NetLogo

NetLogo[14] is a popular simulation modeling environment which can run either natively or in the browser. It uses its own DSL specialized in modeling complex multi-agent systems and is derived from the rich history of Logo variants[12]. Regardless of whether NetLogo is run natively or in the browser, all computation occurs on the CPU. This design choice is powerful enough for many operations, however, the speed at which these simulations run at could be greatly accelerated by moving computations which are able to be parallelized over to the GPU. In order to create NetLogo simulations, a user needs to write NetLogo code in a code editor integrated into their webpage or in the "code" tab of the desktop application. This requires a user to learn the NetLogo DSL and type their simulations out by hand. Since NetLogo has been available since 1998 there are many existing simulation files which developers can find through online sources to use as resources in their own simulations.

To validate and test NetLogo's performance we found an implementation of John Conway's Game of Life[4] (GOL) simulation authored by the creator of NetLogo, Uri Wilensky, in 1998[17]. When evaluating performance of NetLogo's implementation of GOL we observed that NetLogo was able to simulate around 260,000 game of life agents in its default configuration. However, when running the GOL on grids as small as 128 x 128, we observed that NetLogo was not able to update the simulation faster than 4-6 times per second. We were never able to get the simulation to run at a stable frame rate of above 10 frames per second. In the cases where we had thousands of game of life agents on the screen we were able to watch

the updates propagate across the display less than once per second. Our takeaways from exploring the capabilities of NetLogo are:

- The tool is not GPU accelerated.
- Users are required to write NetLogo code in the NetLogo code editor.
- There are memory constraints which are not communicated with the user.
- Models that should benefit from parallel processing are limited by the CPU and run slowly.

However, despite these drawbacks the unique value which NetLogo has is that the tool has been around since 1998 and many simulation models have been developed by the community. This makes it easy for a developer to go online and find existing simulations which they can adapt to their personal needs.

2.1.2 Anylogic

Anylogic[3] is a cross-platform simulation modeling tool popular for business applications. AnyLogic uses a partial node-graph approach to designing simulations. Users are able to drag out specific nodes into their scene and link them together, however they still need to type code into a separate "properties" side-panel to adjust properties of individual nodes. When running an AnyLogic simulation, a user needs to first build the simulation then run the simulation by selecting two different options in a menu panel. Running the simulation opens a separate window which copies the layout and look of the node graph however, now the node graph is interpreted an not editable.

Like NetLogo, AnyLogic is written in Java and uses the JVM to interpret simulations. AnyLogic is not GPU accelerated and relies on a large amount of CPU ram and a CPU with multiple cores to interpret simulations quickly. The AnyLogic documentation states, "We recommend to have 4-8GB of memory and a modern processor with at least 2 cores for optimal performance (more cores benefit pedestrian modeling and experiments with multiple runs)"[13] This implies that there is a level of parallelization being applied to the simulation through multithreading across different CPU cores. To evaluate the performance of AnyLogic we found an implementation of Conway's Game of Life written by AnyLogic themselves. In the both the web simulation viewer, and the native desktop simulation viewer

we were able to run the simulation on a grid of 100 x 100 agents at about 20 FPS maximum. This is significantly better than NetLogo, however it is still drastically below the performance possible by running GOL on the GPU.

AnyLogic allows users to import their own assets to make truly unique and detailed simulations and also has 3D capabilities. When creating agents in AnyLogic a user is able to write code in Java that they would like the agents to execute. This enables users to add their own custom behaviors outside the capabilities the AnyLogic designers initially intended. AnyLogic also has a cloud-based simulation distribution system which enables simulation sharing between individuals. Simulation developers are able to search this system to find simulation files which then could then modify to build their own simulations. Our takeaways from AnyLogic are:

- The tool is flexible and more performant than NetLogo.
- AnyLogic enable users to use a high-level programming language to create simulations which can then be modified with Java to add bepoke functionality.

2.1.3 FlameGPU

FlameGPU[10] is a Python interface for creating GPU accelerated agent-based simulations. In order to get started with FlameGPU, a user needs to ensure that their system has an installation of Python3, CUDA, and a C++ compiler[1]. When investigating FlameGPU we were unable to easily create a working configuration. We found that there were parts of the simulation creation process which requires users to write both Python and C++. We also had consistent issues getting the framework to evaluate anything. We were disappointed by this because the framework is using CUDA, a language created by Nvidia which leverages the full power of Nvidia GPUs[11] which we believed would have led to some of the best performance metrics out of any of the simulation tools we looked into. Our takeaways from FlameGPU is that the project has potential to be powerful, but requires specialized knowledge that is beyond many intermediate to advanced programmers.

[1]See https://docs.flamegpu.com/quickstart/ for details.

2.1.4 GPU-IO

GPU-IO[6] is a web based graphics framework created by Amanda Ghassaei. This framework allows a programmer to compose complex simulations in WebGL by designing a chain of shaders which pass into one-another. The project is written in TypeScript and can be integrated into many different web frameworks.

GPU-IO is published with several example simulations one of which is Conway's Game of Life. We evaluated the performance of GPU-IO through analyzing its execution of the Game of Life. GPU-IO was able to simulate 2560 x 1440 (3,686,400) Game of Life agents at a consistent 144 FPS. We believe that the simulation would be able to execute faster but was limited to the refresh rate of the monitor. GPU-IO was much more performant than any of the other simulation frameworks we investigated, as GPU-IO is GPU accelerated therefore the computations for the simulation logic was performed in parallel on the GPU.

In order to create simulations with GPU-IO, a user needs to write code in TypeScript as well as GLSL. While the performance of GPU-IO was the highest of any of the frameworks we evaluated, we believe that even more performance could be achieved through the use of Compute Shaders. Compute Shaders are specialized programs which can be parallelized in groups to perform the same task across multiple GPU threads very quickly. These types of shaders are excellent for accelerating agent-based simulations where all of the agents are using the same logic. GPU-IO targets WebGL as its graphics backend which is a subset of OpenGLES[1] that does not support Compute Shaders.

GPU-IO is not in active development and the developer has stated that they are pursuing other projects, ".. I'm switching gears a bit to focus on some new projects, but I'll be continuing to use gpu-io as the foundation for almost everything I'm working on."[6] . Our takeaways from GPU-IO is that it is a highly performant framework due to GPU acceleration. The shader chaining idea allows simulations with any number of steps to be created. When creating simulations in GPU-IO the user is required to write both Typescript CPU code and GLSL GPU code, in addition to managing project boilerplate for getting a website that uses the framework up and runnning.

2.2 Graphics Libraries

We investigated several modern graphics APIs to learn what capabilities they offered that may benefit agent-based simulations. We were specifically looking for an API which was cross-platform, executed on GPU hardware, and had any additional capabilities that could be leveraged to accelerate agent-based simulations. Our investigations into different graphics APIs are found below:

2.2.1 WGPU

WGPU[5] is a Rust-based graphics library which transpiles its intermediate representation to many different modern graphics APIs including Vulkan, Metal, and DirectX12[2]. A user can setup a WGPU project, write their code, and then target a specific graphics API of their choice. This enables WGPU programs to run natively on Mac, Linux, Windows, and the web. While WGPU was inticing for a variety of reasons, its dependency on Rust was not a good fit for developers of this project.

2.2.2 Native Vulkan

We considered using the Vulkan[15] Graphics API as the graphics backend for our project. Vulkan is a modern Graphics API which enables a programmer to control exactly what the GPU is doing. The downside of this power and flexibility is that the user needs to meticulously specify everything that they want the GPU to do. To evaluate the feasibility of using Vulkan as the graphics API for our project we set out to create a simple window with a triangle centered on the window. We wanted the window to redraw as fast as possible and maintain the aspect ratio when the window was resized. In order to achieve this we ended up writing about a thousand lines of code to select a GPU, construct a swapchain, and then rebuild the swapchain on window resize. Vulkan is very powerful but the need to be so specific when programming limited the speed at which we could develop Apiary.

2.2.3 OpenGL

Our final investigation was into OpenGL[2]. We looked through the documentation on modern versions of OpenGL and decided that at a mini-

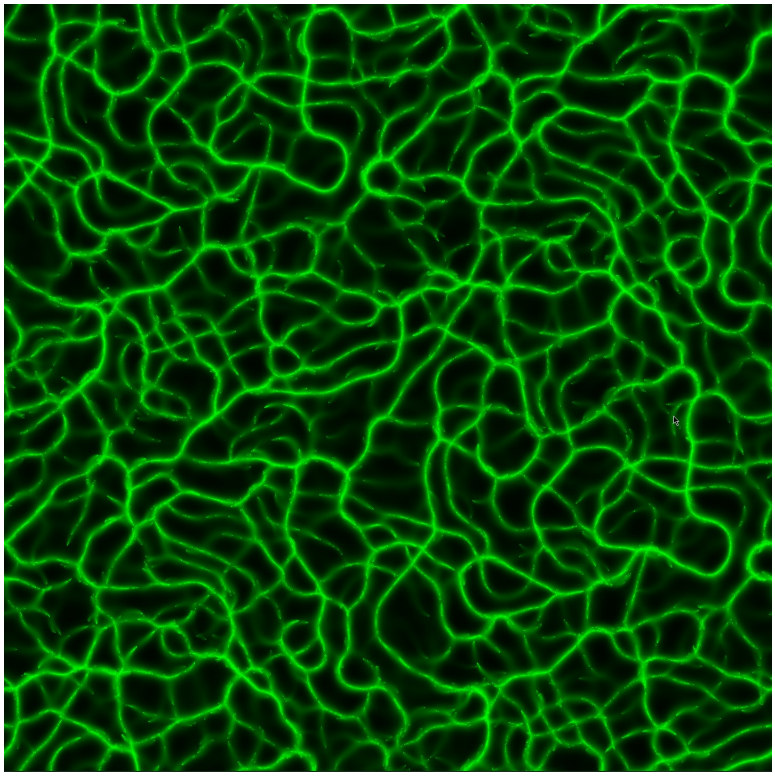
[2]https://github.com/gfx-rs/wgpu

mum we wanted to use OpenGL 4.3, as this version of OpenGL introduced Compute Shaders and Shader Storage Buffer Objects (SSBOs). As stated previously, Compute Shaders are specialized programs which can be parallelized in groups to perform the same task across multiple GPU threads very quickly. These types of shaders are excellent for accelerating agent-based simulations where all of the agents are using the same logic. SSBOs are buffers in GPU memory which can be indexed into as if they were large arrays. A user defines a 'struct' and a number of instances of that 'struct' which they wish to create and asks the GPU to allocate a buffer with that amount of space. This buffer can be accessed from any part of the shader pipeline including Compute Shaders. We use these features together to parallelize execution of tasks where the associated data is stored in GPU memory. This alleviates the need to transfer data between the CPU and GPU with each frame of video, creating higher throughput and framerate.

To evaluate the feasibility of using OpenGL 4.3 we created a test environment and tried to allocate as many agents as possible. We found that the limit as to how many bytes we could address in GPU memory was with a signed 32 bit integer or 2,147,483,648 bytes. This is due to the limitation of how indexing into an array works in GLSL, where it expects a signed integer to be passed as the index to its array dereference function. A user can specify that an SSBO contains instances of a specific 'struct'. This will allow a user to get an instance of that struct from the byte buffer without needing to reconstruct primitives from individual bytes. On an RTX3080 we were able to allocate 130,000,000 16-byte agents and make these agents perform the game of life simulation at around 150 frames per second. This is equivalent to running the game of life on a roughly an 11,400 x 11,400 grid. This performance was superior when compared to the other frameworks we investigated and was the primary justification for choosing to develop with OpenGL.

3 Design and Current Progress

We have been working on this project since September of 2022 and decided to use OpenGL 4.3 in December in 2022. We have been steadily working on our simulation engine since then and have some complex simulations working in our engine such as Physarum[7] simulation which can be seen below.



Physarum simulation in our simulation engine with 10 million agents

We are working towards creating a high-level domain specific language (DSL) for agent-based modeling similar to the blueprinting system used in Unreal Engine. This dataflow language will describe specific aspects of agent-based simulation and will be compiled by Apirary into GLSL. The choice to use a high-level language was made because using a high-level language allows us to vary the target language which our simulation engine uses. In the future we may want to target a different graphics API. To achieve this we could alter the transpilation step for each of the nodes in our language to produce valid code for whichever graphics API we chose to target next. We have started by providing nodes which cover the full GLSL function specification. Any built-in function in GLSL can be represented in our high-level language. We would like for our language to be flexible enough that users could add additional functions to fit their specific needs. If a user wants to add specific code for generating random numbers they should be able to write a small amount of code which then gets processed

by our simulation engine to create new nodes.

3.1 How our DSL will be interpreted

We have implemented a stack machine to transpile our Intermediate Representation (IR) to GLSL, based on ideas found in Wakefield and Roberts' research on virtual machines for audio language design[16]. Our IR is comprised of a list of comma delimited strings where each string is a piece of data or a function directive. A simple example of a function directive would be "@add" mapping back to an internal representation like this, "2 , (param[0] + param[1])". This function directive registers the "@add" string with 2 elements and the transpilation function of "(param[0] + param[1])" when this function directive is interpreted it will pop two elements off of the object stack, then substitute them into the transpilation function and push the result back onto the object stack. This process continues until the evaluation stack is exhausted and the object stack has be transformed into a single element representing the GLSL equivalent of the initial IR.

To date, we have implemented the full GLSL 4.3 function specification in function directives. We additionally provide a public functional interface to allow developers to integrate with this function directive system in order to register their own custom directives. The intent behind this design decision is to enable a community to develop additional functionality or modifications to our IR system.

Below we show a simple example of how our stack machine transpiles our IR to GLSL.

Input: [3, 1, 2, "@add", "@mul"]
Transpilation Steps

3

3, 1

3, 1, 2

3, 1, 2, @add

3, (1 + 2)

3, (1 + 2), @mul

(3 * (1 + 2))

The result of running our stack based transpiler produces the following GLSL form the initial IR (3 * (1 + 2)).

3.2 Planned Simulations

We would like to evaluate the flexibility of our framework by implementing several famous simulations. To date, we have implemented Conway's Game of Life, and Physarum. We will add a Boids simulation, a 2D pressure/flow simulation, and other common representative simulations found in agent-based systems. When looking for research that had already been conducted in this area we came across, "A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units"[8]. In this paper the authors create an agent-based simulation framework and choose to implement and use StupidModel[9] as a benchmarking metric. We will compare their results with StupidModel running in Apiary.

4 Schedule

When initially planning out this project we created the schedule found below. We have kept to this schedule which denotes the preliminary research we conducted and the experiments we did to see how performant we were able to make our own Simulation Engine.

- September / 2022: Preliminary Research
  - Collaborate with simulation modelers to learn what tools they use and what features are most useful to them.
  - Find modern simulation frameworks and investigate them.
  - Explore modern graphics APIs and frameworks to determine what would best facilitate our project goals.
- October / 2022: Experimentation
  - Build small test program in Rust with WGPU.
  - Use GPU-IO to setup a React+WebGL project to evaluate performance.
- November / 2022: Project Setup
  - Build small test program in Native Vulkan.

  - Decide on Intermediate Representation (IR) format.

- December / 2022: First Simulations
  - Build small test program in GPU-IO
  - Build a Java Application with OpenGL 4.3 to test how compute shaders interface with other parts of the shader pipeline.

- January / 2023: High-Level Language
  - Implement transpiler to convert from IR to OpenGL Shader Language (GLSL).
  - Program Conway's Game of Life in our IR.
  - Evaluate how many agents our simulation engine can represent.
  - Test simulation engine on multiple hardware configurations.
  - Integrate ImGUI with the simulation.
  - Create interface for a "node" which can then be transpiled to GLSL
  - Create Nodes for common functions which users want to do in simulations such as math functions, vector operations, and reading and writing to agents.

- February / 2023: Proposal
  - Present Proposal.
  - Continue to develop the high level language representation of simulations.
  - Implement transpiler from high-level language to IR to GLSL.
  - Recreate Conway's Game of Life and Physarum in high-level language.

- March / 2023: Complex Simulations and User Experience
  - Experiment with 3D simulations.
  - Create save file format for high-level representation of simulations.
  - Add support for real-time adjustment of uniform variables and control of the simulation.

- April / 2023: Results

- Create Performance charts showing rough performance metrics based on system hardware.
- Create example simulations which users can take from when making their own simulations.
- Document all nodes in the language and create tutorial for how to use our simulation tool.
- Submit Thesis report
- Presentation of Thesis
- Publish Website where users can go to download and use our Simulation Framework.

References

[1] URL: https://registry.khronos.org/OpenGL-Refpages/es3/.

[2] URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.30.pdf.

[3] Andrei Borshchev, Yuri Karpov, and Vladimir Kharitonov. "Distributed simulation of hybrid systems with AnyLogic and HLA". In: *Future Generation Computer Systems* 18.6 (2002), pp. 829–839.

[4] Martin Gardner. "The fantastic combinations of John Conway's new solitaire game "life"". In: *Scientific American* (Oct. 1970), pp. 120–123. URL: https://www.ibiblio.org/lifepatterns/october1970.html.

[5] Gfx-Rs. *Gfx-rs/WGPU: Safe and portable GPU abstraction in Rust, implementing webgpu API.* URL: https://github.com/gfx-rs/wgpu.

[6] Amanda Ghassaei. *Amandaghassaei/GPU-IO: A GPU-accelerated computing library for physics simulations and other mathematical calculations.* URL: https://github.com/amandaghassaei/gpu-io.

[7] Jeff Jones. "Characteristics of pattern formation and evolution in approximations of physarum transport networks". In: *Artificial Life* 16 (2 2010), pp. 127–153. ISSN: 1064-5462. DOI: 10.1162/artl.2010.16.2.16202. URL: https://uwe-repository.worktribe.com/output/980579.

[8] Mikola Lysenko and Roshan M. D'Souza. "A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units". In: *Journal of Artificial Societies and Social Simulation* 11.4 (2008), p. 10. ISSN: 1460-7425. URL: https://www.jasss.org/11/4/10.html.

[9] Steve Railsback, Steve Lytinen, and Volker Grimm. "StupidModel and extensions: A template and teaching tool for agent-based modeling platforms". In: *Swarm Development Group. http://condor. depaul. edu/slytinen/abm* (2005).

[10] Paul Richmond et al. "High performance cellular level agent-based simulation with FLAME for the GPU". In: *Briefings in Bioinformatics* 11.3 (Feb. 2010), pp. 334–347. ISSN: 1467-5463. DOI: 10.1093/bib/bbp073. eprint: https://academic.oup.com/bib/article-pdf/11/3/334/606876/bbp073.pdf. URL: https://doi.org/10.1093/bib/bbp073.

[11] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[12] Cynthia Solomon et al. "History of logo". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), pp. 1–66.

[13] *System requirements*. URL: https://anylogic.help/anylogic/ui/system-requirements.html.

[14] Seth Tisue and Uri Wilensky. "Netlogo: A simple environment for modeling complexity". In: *International conference on complex systems*. Vol. 21. Citeseer. 2004, pp. 16–21.

[15] Vulkan. *Home: Vulkan: Cross Platform 3D Graphics*. Dec. 2022. URL: https://www.vulkan.org/.

[16] Graham Wakefield and Charles Roberts. "A virtual machine for live coding language design." In: *Proceedings of the New Interfaces for Musical Expression Conference*. 2017, pp. 275–278.

[17] Uri Wilensky. *NetLogo Life model, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL*. 1998. URL: http://ccl.northwestern.edu/netlogo/models/Life.