Controlling Outbreak Rates of Novel Ailments (CORONA) with GPU Accelerated Simulation

Taylor Chatfield
Department of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, IN, USA
chatfitt@rose-hulman.edu

Leela Pakanati
Department of Electrical and Computer Engineering
Rose-Hulman Institute of Technology
Terre Haute, IN, USA
pakanalk@rose-hulman.edu

***Abstract*—In times of global pandemic, simulating the way a disease spreads within a population can provide key data for agencies attempting to come up with policies to control the spread and mitigate the effect. In this paper, we develop a simulation based off of existing Susceptible-Infected-Removed (SIR) simulations focused on tracking disease spread as people move between locations. We then use a GPU to accelerate this simulation utilizing Nvidia's CUDA tools. With various additional GPU optimizations, we were able to achieve a total 2.45x speedup on the GPU implementation over the baseline CPU implementation.**

***Index Terms*—coronavirus, simulation, disease, epidemic, GPU, CUDA, parallelism**

I. Introduction

As of writing, the world is currently in the midst of a COVID-19 pandemic. This is not the first pandemic seen, and it is unlikely to be the last. Preparation for possible future pandemics requires the ability to both invent and test preventative measures to limit the spread of a disease. Simulations provide this testing capability by simulating virtual pandemics orders of magnitude faster than real time and allowing broad levels of variability. This paper does not analyze the spread of a disease or draw any conclusions for prevention measures, but looks to provide a GPU-accelerated platform with which those experiments may be conducted.

Many groups have created simulations, ranging from simple random walk simulations that are meant to educate the public on preventative measures [1] to grid-based simulations of cities to see where 'hot spots' of high transmission rates may occur [2]. More detailed analyses also exist, albeit still with a public awareness goal instead of scientific research, that take into account the divisions of a geographic location into communities that have higher internal transmission rates [3]. These simulations approach realistic situations, but do not accurately reflect the movement patterns of a populace, and therefore, may not be providing the most accurate simulations.

Nvidia's CUDA platform allows GPUs to be used for other uses than graphics and video. GPUs are highly specialized to perform well on high-throughput loads and are well-suited to running massively parallel programs. Random-walk simulations may be suited to parallelism, but their inaccuracy is undesirable. More independent movement in simulations may allow more opportunity to exploit parallelism as there is less dependence on other entities' data. Part of this paper's objective is to determine whether the GPU can be fully utilized to accelerate simulations of this type. With this CUDA platform, many different optimizations are present to try to achieve higher throughput, and therefore, quicker simulations.

II. Simulation Overview

This simulation, in contrast to the examples given, uses a location-based model. Individual people in the simulation exist and intermingle randomly at each location, and have the ability to move between different locations. We believe that this model is more accurate to real life, as people tend to travel point-to-point and intermingle at those locations, such as restaurants, offices, schools, and homes.

Individual people in the situation are given one of five states, following the S/C/I/R/D model. Susceptible (S) people have not contracted the disease and are able to become infected. Carrier (C) and Infected (I) people are actively infected with the disease, but carriers do not show symptoms. Recovered (R) people have been infected but are both no longer infected and unable to become infected again. Deceased (D) people have died from the disease and are no longer relevant to the simulation.

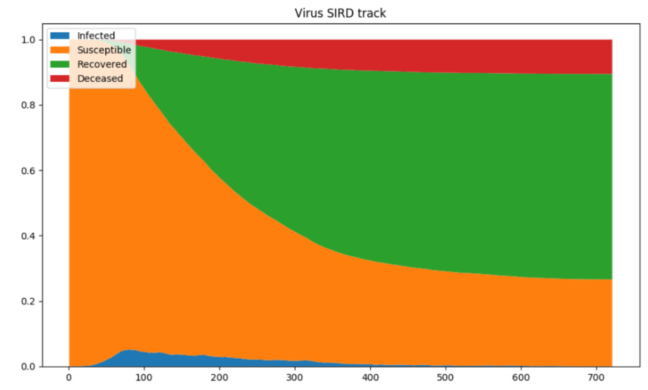

Fig. 1. Example graph generated from simulation data.

Epidemics in the situation can be customized by defining a population, locations, and disease specifics such as its spread factor, fatality rate, and average infection duration. These factors are custom to the simulation and may not correspond to medical standards or terminology due to our inexperience with infectious diseases and the extra challenges required to properly implement them.

Statistics are collected after every infection step, but a very granular level. The number of people in each category (with the exception of carrier status population, which is represented as infected) is recorded and written out to the console. This can be used to construct visualizations such as the graph shown in Figure 1.

III. Single-Threaded Algorithm

The core loop of the single-threaded implementation is split into four phases:

- Spread Disease
- Collect Statistics
- Determine Movement
- Execute Movement

In the spread disease step, the simulation loops through each location and determines if an infected individual exists. If so, it will randomly infect other people in that location based on the values provided in the configuration file. The collect statistics step loops through all locations and records the number of people of each infection status. In determine movement, the simulation loops through each location and determines whether each individual there is expected to move. If the individual is expected to move, they are added to a separate list for that location, and if not, they are added to the separate list at their current location. After this has completed (to prevent concurrency issues), these lists' contents are moved into the population list at each location. Since only a single thread is used, this means that each time step, that thread must visit each person and perform calculations four times. An overview of this algorithm is shown in Figure 2.

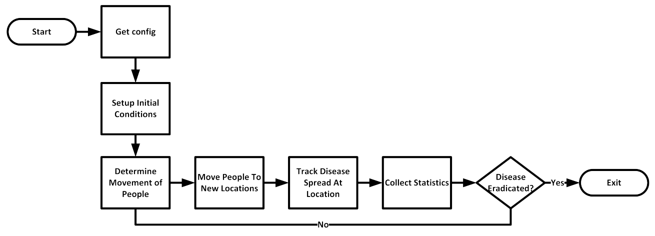
Start
Get config
Setup Initial Conditions
Determine Movement of People
Move People To New Locations
Track Disease Spread At Location
Collect Statistics
Disease Eradicated?
Yes
Exit
No

Fig. 2. CPU Code Flowchart.

IV. Parallel Algorithm

A. Overview

The main objective of parallelization was to reduce the time spent calculating movement and whether or not an individual was exposed to the disease. We did this by turning every individual in the simulation into their own thread, organized into blocks by their location. This made the most sense, since individuals only interact with others in their same location during each time step. However, since there is no way to predict the number of people in a location at any given time, the block size was determined to be the maximum number of people in any location (which is set statically in the program, unable to be modified by the configuration file). This causes the kernel to allocate significantly more threads than it needs. For example, if locations can hold 256 people, and a population is defined as 200 times the number of locations, an average of 56 threads per block will be unused. However, trying to minimize this overhead will cause extreme slowdowns and control divergence as threads struggle to find open spaces for their respective people to attempt to switch locations. The single-threaded implementation also originally used classes and more complex data types, but these pointers were removed to prevent memory indirection and spatially correlate all the data that was being used.

The new loop structure added additional steps to further segment functions:

- Advance Infection
- Spread Disease
- Collect Statistics
- Combine Statistics
- Determine Movement
- Execute Movement

The spread disease was split into advance infection, which updates the status of people already infected with the disease, and spread disease, which spreads the disease to other people in the same location. Collect statistics was split into collect statistics, where the thread determines the status of its corresponding person, and combine statistics, where these individual tallies are combined together from the various threads to get their final values. Figure 3 shows an overview of the updated algorithm.

Each section below will describe an optimization that was attempted and compare it to the previous iteration to investigate whether the optimization was effective. All data in the following sections were generated on an AMD Ryzen 9-3900X CPU with an Nvidia GTX1060 3GB GPU and averaged over 20 runs with the same simulation parameters (population=15000, locations=400).

B. Initial Parallel Implementation

The first iteration of a parallel implementation saw three of the four functions of the core loop migrated to the GPU, with the exception of the determine movement step. This function was initially separated from the determine movement step to prevent concurrent modification issues, and was not migrated to the GPU for the same reasons. CuRand had to be used instead of the C++ random libraries. We did not expect any speedup from this original implementation, and our results confirmed that the initial parallel implementation was almost 95 times slower than the single-threaded implementation, as shown in Table I.

We expected the source of the slowdown to be the data transfers between the CPU and GPU that were occurring every time step. However, performance analysis using NvProf indicated that the spread disease and determine movement

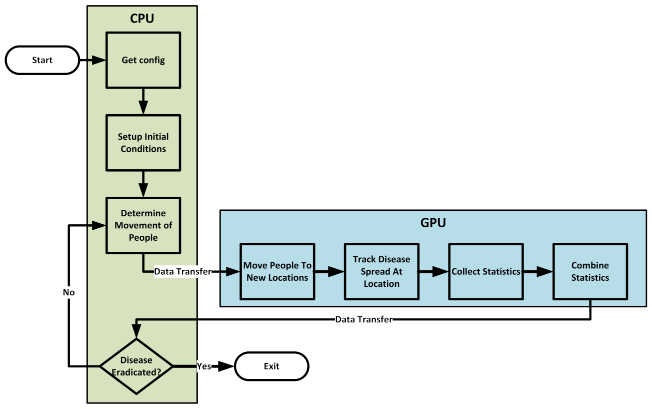

Fig. 3. GPU Initial Code Flowchart.

steps were taking roughly 150 times as long as the memory copies.

C. CuRand Initialization Problem

These functions used random number generation, unlike the third function. We discovered that the functions were initializing a random number generator every run. This issue was resolved by changing the randomization to initialize once before any of the core loop functions were called. While causing a significant reduction in runtime, this was less of an optimization and more of a correction for an implemented error. As shown in Table I, the parallel implementation was still almost 4.3 times slower than the single-threaded implementation.

TABLE I
INITIAL PARALLELIZATION LATENCY AND SPEEDUP

Implementation	*Advance Infection*	*Spread Disease*	*Data Transfer*	*Total Runtime*	*Total Slowdown*
Single-Thread	88.15us		N/A	0.291s	1
Initial Parallel	15590us	14890us	1.0ms	27.564s	(94.72)
CuRand Init. Fix	5.953us	52.70us	1.0ms	1.244s	(4.275)

D. Pinned Memory

The main bottleneck of the parallel implementation was now the data transfer, as we originally expected. To attempt to minimize the duration of these transfers, we implemented the host array in pinned memory as this is expected to reduce overhead for memory copying between the CPU host and GPU device. There was insignificant speedup that produced no real benefit, as shown in Table II. Regardless, it remained implemented as there did not appear to be any drawbacks.

E. Complete Parallelization

The final step for completely implementing parallelization was to remove the data transfer between the CPU and GPU during the core simulation loop. The first method attempted

TABLE II
PINNED MEMORY FUNCTION LATENCY AND SPEEDUP

Implementation	*Data Transfer*	*Total Runtime*	*Total Slowdown*
W/o Pinned memory	1001us	1.244s	(4.275)
W/ Pinned Memory	997us	1.232s	(4.233)

was to implement the single-threaded execute movement function as a single-threaded function on the GPU. As shown in Table III, this caused the program to run even slower than before at 27 times slower than the single-threaded implementation.

The primary limitation of implementing parallelization on this function was that each thread contained an individual that was going to attempt to move to an array that any number of threads could also be trying to write to. By using an atomic operation to obtain an index to write its data to, each thread can concurrently write to these arrays. After implementing this, Table III shows that the parallel implementation became 1.2x faster than the single-threaded implementation.

To further improve performance of this function, since the the people not moving locations, a majority, would be staying within the same block, we utilized a shared memory array to keep track of them. This array is then copied into the global people array for the location at the end of the function's execution. This did not cause the latency of the function to decrease by much, but still resulted in a significant runtime decrease due to the lower amount of memory accesses. As shown in Table III, these changes increased the parallel implementation's speed to execute more than 1.4x faster than the single-threaded implementation.

TABLE III
PARALLELIZING DETERMINE MOVEMENT

Implementation	*Data Transfer*	*Determine Movement*	*Total Runtime*	*Total Speedup*
Single-Threaded On GPU	1001us	12478us	7.969s	.0365
Parallelized	4.54us	30.551	0.242s	1.202
Parallel w/ Shared Memory	4.023us	29.917us	0.206s	1.412

V. FURTHER PARALLEL IMPROVEMENTS

A. Optimizing Location Sick Person Count

Another function that we worked at optimizing was the spread disease function. The first necessary calculation is to determine the number of infected people in a given location. We determined two methods of optimization, one using reduction and one using atomic operations. The reduction method had each thread write a 0 or 1 to its thread index in an array to indicate whether the corresponding person was infected. From there, this array was reduced by adding all the elements together. The atomic method had each thread increment a shared memory variable if the corresponding person was infected.

The atomic function was found to run faster, taking an average of 6.141us compared to the shared memory function taking 11.67us, detailed in Table IV. Both of these were significant improvements over the previous 49.02us average runtime of the spread disease function. We proceeded forward with the atomic strategy for further development.

TABLE IV
OPTIMIZING SICK PERSON COUNT

Implementation	*Spread Disease*	*Total Runtime*	*Total Speedup*
Parallel w/ Shared Memory	49.016us	206ms	1.412
Strategy: Reduction	11.673us	185ms	1.573
Strategy: Atomic	6.141us	168ms	1.732

B. Optimizing Statistics

Collect statistics originally had each thread count the number of people in each S/I/R/D category. This used a switch statement, and modified a value in an array that was the size of the maximum population in a location times the number of locations. During the combine statistics phase, these values would get reduced and accumulated using the thrust library. This method was slow and could be optimized using atomic counters using shared memory variables. The atomic memory strategy has each of these four categories track a single value that each thread increments. Removing control divergence can also be accomplished by writing the result of a boolean check (which evaluates to 0 or 1) to all four categories; this quadrupled the number of atomic add operations. Both of these methods reduced the collect statistics and statistics reduction functions as shown in Table V but the one tried to reduce control divergence by increasing the number of atomic operations did slightly worse with a 8.455us collect statistics runtime versus 7.98us for keeping the control divergence. With this atomic counters method, we were able to achieve a total 1.94 speedup. These atomic counters with the control divergence still included was used for further development.

Further, we also implemented our own reduction method instead of using the thrust library, to verify the efficiency of the library. The results in Table VI show while the reduction function itself appears to be taking more time, the overall runtime was still reduced, providing an overall speedup of 2.19. We suspect this is due to the thrust library being called from within the host code, resulting in overhead not shown in this data.

With this custom reduction kernel, we were now able to store the generated statistics on the device rather than copy them to the host CPU and print out to the console on every iteration. We stored each iterations statistics output on the device, only transferring single variable to check whether the simulation should continue, and finally copying the full statistics to host device and printing all at once only at the end. This allows for a copy single large data transfer and print in place of a multitude of small ones. As shown in Table VI, this reduced the amortized data transfer time to 2.23us, bringing the complete speedup to 2.45x.

TABLE V
OPTIMIZING STATISTICS COLLECTION

Implementation	*Collect Stats*	*Stats Reduction*	*Total Runtime*	*Total Speedup*
Parallel W/ Sick Count Opt.	12.015us	17.834us	168ms	1.732
Atomic Counters w/ Control Div.	7.98us	4.92us	150ms	1.940
Atomic Counters w/o Control Div.	8.455us	4.91us	157ms	1.854

TABLE VI
OPTIMIZING STATISTICS COMBINATION

Implementation	*Stats Reduction*	*Data Transfer*	*Total Runtime*	*Total Speedup*
Parallel w/ Thrust Reduction	4.91us	4.894us	150ms	1.940
Custom Reduction	8.167us	4.792us	133ms	2.188
Custom Reduction w/ Consolidated Stats	5.635us	2.23us	119ms	2.455

This final GPU imlpementation is outined by the flowchart in Figure 4.

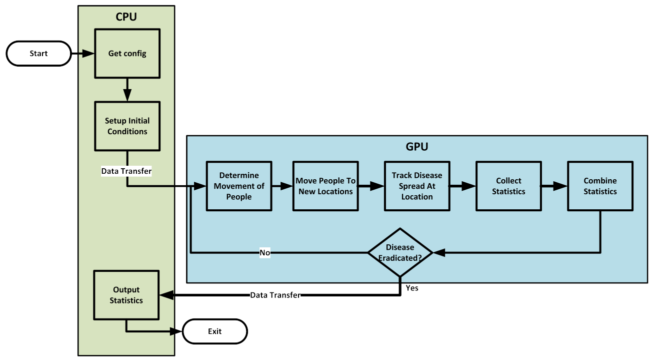

Fig. 4. GPU Final Flowchart.

VI. FUTURE WORK

A. GPU Optimizations

On the GPU, the primary function with the longest runtime that wasn't further optimized was the Execute Movement function. The majority of work in this function involves a memcpy on the device across its global memory. It may be possible to improve the performance of this memory operation. Further, all of the individual functions are currently called from the host code. This was useful for profiling by function; however, further speedup may be realized by moving this to a single kernel function with individual device functions.

B. Simulation Extension

The simulation was set up to be easily extended to allow for better representation of a population. This can be done by assigning location types and relation to individuals, e.g. home, office, preferred store. Further, locations can be assigned individual probabilities for people leaving to better represent these types of locations, e.g. homes would have a lower probability of leaving than stores.

The main goal of these types of simulations is to test the effectiveness of different policies being applied to the model. Simulation testing with these types of policies can be implemented on top of the existing simulation to test things like implementing quarantines, social distancing, travel bans, etc.

VII. Conclusion

Compared to the initial single-threaded implementation, the parallel implementation and optimizations yielded a 2.45 times speedup. This simulation is not particularly mathematically intensive, but contains significant control complexity. Due to this nature, a multi-threaded CPU implementation would also be worth exploring and may be a better fit for this workload. With this new tool, the world is ready to face the next pandemic **if** we ever get through the current one.

References

[1] H. Stevens, ”Why outbreaks like coronavirus spread exponentially, and how to ’flatten the curve’,” *The Washington Post*, Mar. 14, 2020, Accessed on: Mar. 30, 2020. [Online].

[2] G. Yeghikyan, ”Modelling the coronavirus epidemic in a city with Python,” *Medium: Towards Data Science*, Feb. 4, 2020, Accessed on: Mar. 30, 2020. [Online].

[3] 3Blue1Brown, ”Simulating an epidemic,” *YouTube*, Mar. 27, 2020 [Video File]. Available: https://www.youtube.com/watch?v=gxAaO2rsdIs. [Accessed: Mar. 31, 2020]