CS267 Final Project

Parallelizing Agent-Based Disease-Spread Models for High-Fidelity Simulations

Group 12

Maya Horii
mjhorii@berkeley.edu

Brian Howell
bhowell@berkeley.edu

Reece D. Huff
rdhuff@berkeley.edu

CS267: Applications of Parallel Computers
Professor James Demmel
Department of Electrical Engineering and Computer Science
University of California, Berkeley

Contents

Introduction

Accurate disease-spread modeling is critical for developing effective mitigation efforts during an outbreak. The most basic epidemiological models use ordinary differential equation representations of the evolution of susceptible, infected, and recovered populations over time, but these models are limited by their simplistic and deterministic nature. In contrast, agent-based models assign attributes to each individual in a population. These attributes, in addition to global rules for movement and interaction between agents, define an individual's behavior patterns. Agent-based epidemiological models are stochastic and can be highly realistic, as the level of detail incorporated from demographics, city geography, and data on time personal usage is arbitrary.

Agent-based models (ABMs) typically loop through each agent at each time step, then evaluate disease-spread progression at their new locations, which becomes prohibitively slow with large population sizes. Because of this, parallelism is required to make high-fidelity simulations and large-scale data generation feasible.

Background Information: Key Prior Work

Collier et al. created the Chicago Social Interaction Model (chiSIM), an agent-based model containing approximately 2.9 million agents [1]. An unparallelized version of this code took around 60 compute hours to run a 10-year simulation. They first implemented parallelization using multi-threading in `OpenMP`. The agent movement loop required a section to be within an `omp critical` directive so that agents were not written to the same place concurrently, which limited speed-up. They then parallelized the model across multiple processes. Because the attributes of a place, as well as the disease status of other agents at a given place, must be known to determine spread, information about places and co-located agents must be known across all processes, or agents must move to the processor containing their next place. The authors decided to go with the second option, where agents move processes based on their in-simulation location. To reduce the amount of agents' movement between processes, they created a network graph of all places based on each agent's individual attributes. It was necessary to ensure that the network graph minimized movement between processes, but also didn't place too many agents in a single process, which caused slowdowns in initial testing. Ultimately, combining multi-threading across multiple processes, they reduced the run time of a 10 year simulation *from 60 hours to 4 hours*, which is much more reasonable. However, to calibrate a stochastic agent-based model, often through a Markov chain Monte Carlo or approximate Bayesian computation process, it must be run many times, and this reduced run time may still be prohibitively high when considering calibration over many parameter dimensions.

Other studies have tackled this problem as well, for instance, Ozik et al. recently studied the spread of COVID-19 in Chicago using a similar model as Collier et al. [2]. On the surface, their simulations appear to be much faster, but major differences in the computer used as well as small differences in the model make their relative performance difficult to compare. Their system was built using `Swift/T`, a language used for MPI programming. They incorporated function calls to both `R` and `Python` functions within the implementation.

The FLAME GPU framework was built to take advantage of GPUs for agent-based modeling [3]. It uses an indirect messaging system to send information from one agent to all other agents, or agents within its local vicinity, or send a information to a specific location. They use stream compaction to avoid sparse data when agents die and are no longer required to be stored in memory.

Hypothesis

We expect that we can significantly reduce the computational time of an existing agent-based disease-spread model using a popular parallel computing framework: `CUDA` with NVIDIA A100 GPU's.

Methodology

The code used in this project was based on an existing agent-based disease spread model. In this model, agents represent individuals in a population, and move in a random direction with a random velocity bounded by a mobility parameter at each time step. The agents within a sub-population are constrained to a 2-d rectangular domain, and multiple sub-populations can exist within the model. Each agent has a status of susceptible, exposed, infected, or dead/recovered. A susceptible agent can become exposed if it is within the "radius of infection" of an infected agent. Then, the exposed agent will become infected after an *incubation period*, and will then become dead/recovered after an *infection period*. The lengths of the incubation and infection periods are different for each agent, and are sampled from gamma distributions.

Agent data structure

Agent characteristics are stored in an array of `struct`'s (which is kept on the GPU while the simulation is running). The `struct` definition is shown below; it is similar to the data structure used for particles in Homework 2 [4]. It contains some added features, including **subPop**, which keeps track of the current sub-population; **original_subPop**, which stores the ID of the sub-population the agent was initialized in; **E_time** and **I_time**, which track the exposure and infection time respectively; timer, which keeps track of either how long an agent has been infected, or how long an agent has been exposed, depending on the current status; **status**, which tracks whether an agent is susceptible (S), exposed (E), infected (I), or dead/recovered (DR); subSim, which tracks which sub-simulation an agent is in; and `id_in_subSim`, which gives an agent an ID corresponding to the agent's place within the sub-simulation. The values of `id_in_subSim` range from (0, (number of agents in the **subSim**)) for each sub-simulation, and are needed to ensure consistent results from a specified random seed regardless of what other sub-simulations are being concurrently run.

```
// Agent Data Structure
typedef struct agent {
    int id; // Agent ID
    double x; // Position X
    double y; // Position Y
    int subPop; //Subpopulation ID
    int original_subPop; //Subpopulation ID of original location
    double E_time; //Exposure time
    double I_time; //Infection time
    double timer; //Current timer (either of infection or exposure depending on disease
       status)
    int status; //0 = S, 1 = E, 2 = I, 3 = R
    int subSim; //subsimulation ID
    int id_in_subSim; //ID of agent (sub simulation specific)
} agent;
```

Random number generation

To ensure consistency, each agent is assigned a random number generator. They are initialized using `curand_init`, which takes a seed and a sequence to specify its state. The seed is set to the sub-simulation's seed, which is specified in the population information `csv` passed as input to the program (see Table 3). The sequence is set to the agent's `id_in_subSim`. This guarantees that a sub-simulation run with a given random seed will always get the same results from that random seed (given all other parameters remain equal).

Agent movement

Agent movement was parallelized by creating a CUDA kernel function to compute all movement, allowing agents to be moved concurrently.

At each time step, an agent moves randomly. The direction and velocity of movement is determined randomly, with the maximum length of movement bounded by the mobility parameter μ. Starting from a position (x_0, y_0), an agent would move to a new location (x_1, y_1) according to:

$$(x_1, y_1) = (x_0 + \beta_x * \mu * \Delta t, y_0 + \beta_y * \mu * \Delta t)$$

where β_x and β_y are independent random variables sampled from a uniform distribution between -1, 1. We used the bouncing mechanism from Homework 2 to prevent agents from leaving their sub-population's domain [4].

In addition to movement within the sub-population, agents may also jump outside of their sub-population. At each time step, an agent has a probability equal to the `jumping_probability` parameter of moving to a different sub-population (specified in the input `csv` file, see Table 3). Agents cannot move outside of their sub-simulation, so jumping is only possible if there are multiple sub-populations within a sub-simulation. If an agent does jump, they uniformly randomly choose one of the other sub-populations within their sub-simulation, and then are placed at a random location within that sub-simulation's domain. To facilitate this, an array was created containing the partial sums of the number of sub-populations per sub-simulation (`cum_ms_per_subsim_d`) so that we are able to determine which sub-populations are valid choices to jump to based on an agent's **subSim** ID. Additionally, we need an array to store the possible choices to jump to, `choices_d`. For simplicity, we made `choices_d` have length equal to (maximum number of sub-populations in any sub-simulation)$\times$(number of agents), so that there would be enough space to store all possible choices for each agent. This could be redesigned to be more memory efficient, partly by utilizing a random number generator to choose an index from an agent's section of `choices_d`.

Updating timers

Any agent that is currently exposed or infected must have their timer updated by adding the time step Δt. This is done in a CUDA kernel which is run for every agent, but only updates timers for agents with status 1 or 2 (exposed or infected respectively).

Updating agent disease status

First, we check if any exposed agents have passed their exposure time (defined in their agent `struct`) and should become infected, or if any infected agents have passed their infection time and should become dead/recovered. This is done in a CUDA kernel, and is easily parallelized.

Second, we check if any infected agents are within the infection radius of any susceptible agents. For each infected agent, we loop over all other agents within their sub-simulation, and check distances if any of the other agents are susceptible. If they are within the infection radius, we update the status of the susceptible agent using an `atomicCAS` (compare and swap) operation, which checks if the susceptible agent's status is still 0 (susceptible), and if it is, replaces it with 1 (exposed).

Tracking data

At each time step, we count the number of agents that have each status per sub-population (they are tracked by their original sub-population, not their current sub-population, though this could be changed in the future) and store it in a matrix of size (number of sub-populations, number of time steps + 1). Each status has one matrix to store this information in. On the GPU, these matrices are stored as 1D arrays. In a CUDA kernel which operates over every agent, we use `atomicAdd` operations to add to the appropriate time step and sub-population slot for each status.

Saving data

At the last time step, we use `cudaMemcpy` to copy over the data tracked at each time step, then store it to a `csv`. If rendering data is requested (see Table 2), the full array of agent information is copied over according to the specified frequency of saving.

Opportunities for improvement

Timer updates, updating disease statuses based on timers, and checking for new exposures do not need to be done for every agent at every time step. They could instead be run for a smaller subset (e.g., only those agents that are currently infected or exposed for timer updates). This could potentially speed up runtimes, however, it would add significant complexity to the code, as at each time step, we would need to check agent statuses, create a sorted list (similar to binning strategy in HW2.3), and keep track of the number of agents in each status. There is also a barrier required between each of these operations, so one agent still could not continue onto the next kernel if the previous kernel did not apply to their status. For these reasons, we chose not to implement this, and instead ran each kernel over all agents.

Since dead/recovered agents do not interact with the rest of the agents, it is not technically necessary to continue tracking or moving them. They could be removed to save memory and computational time.

Finally, we did not implement spatial binning. This would be useful for sub-simulations with large numbers of agents.

Results

Strong scaling with a single sub-simulation

We tested the strong scaling of the parallelized GPU code in comparison to the serial CPU code. Here, we run a single sub-simulation, in which the total population varies (the domain varies along with it to hold the density of agents constant at 10000 agents per square length unit), and the rest of the parameters are defined as seen in Table 1. As seen in Figure 1 (tests run on Perlmutter), the serial CPU code performs better than the CUDA GPU code at very small population sizes – at a small scale, the lower speed GPU processors are less efficient than the CPU. However, as the population size increases, the GPU code quickly becomes much faster. The log-log slope of the serial CPU code is approximately 1.5, which is due to the fact that infected agents must check all other agents within their sub-simulation to see if they are 1.) susceptible, and if so, if they are 2.) within the infection radius of the infected agent. The overall log-log slope of the CUDA GPU graph is 0.81, but the slope at the end of the graph is around 0.87. This is likely because the number of agents is still too low to fully utilize the maximum number of threads available in the GPU – we expect that at full capacity, the GPU code slope would match the CPU code. However, the runtimes were too slow to continue testing at higher numbers of agents.

Argument	Type	Description	Example
Total population	`int`	Total number of agents within a sub-population	varies
`S`	`double`	Percentage of **Total population** that is *susceptible*	0.99
`E`	`double`	Percentage of **Total population** that is *exposed*	0
`I`	`double`	Percentage of **Total population** that is *infected*	0.01
`R`	`double`	Percentage of **Total population** that is *recovered*	0
`X_neg_lim`	`double`	Lower *x* bound of agents	0
`X_pos_lim`	`double`	Upper *x* bound of agents	varies
`Y_neg_lim`	`double`	Lower *y* bound of agents	0
`Y_pos_lim`	`double`	Upper *y* bound of agents	varies
Sub-simulation	`int`	Sub-simulation ID	1
Seed	`int`	Random seed for the sub-simulation	0
M	`int`	Total number of sub-populations for that particular sub-simulation	1
Mobility	`double`	Length of vector that agents jump in a second	0.01
Jumping prob	`double`	Probability that an agent jumps to a new sub-population	0

Table 1: Arguments within input `csv` file

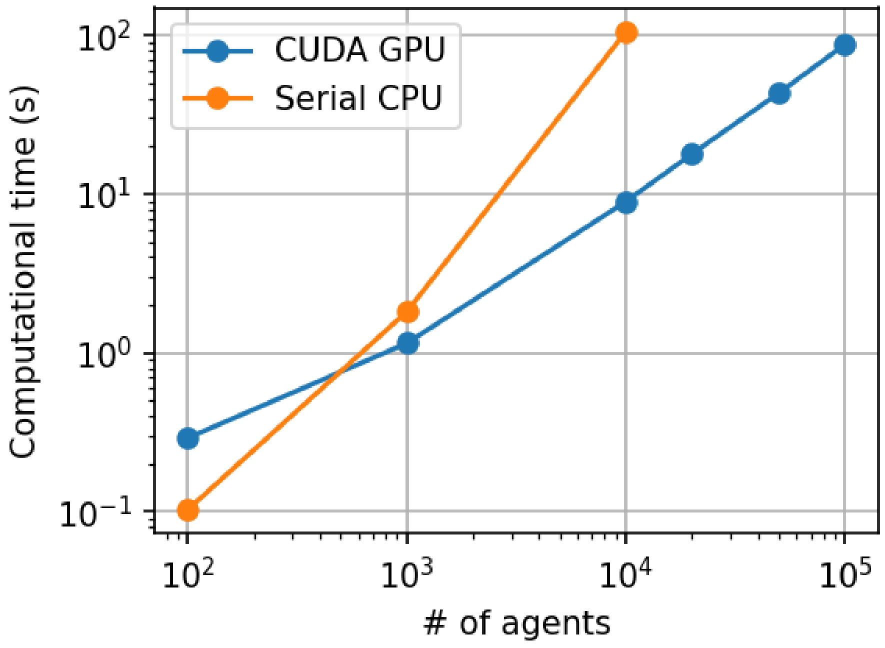


Figure 1: Computational time of GPU code as the number of agents varies in a single sub-simulation with `NUM_THREADS = 256`.

Strong scaling with multiple sub-simulations

Next, we tested the strong scaling, but increased the number of agents proportionally to the number of sub-simulations. We used the same parameters as in Table 1, but held total population constant at 100, and added additional sub-simulations to match the desired total number of agents. The results (tested on Perlmutter) are shown in Figure 2. Again, we see that at very small numbers of agents, CPU code outperforms GPU code, but GPU code quickly overtakes it as agents increase. The log-log slope of the CPU code is ≈ 0.81. The overall GPU code log-log slope is around 0.33 due to GPU under-utilization, but the slope at the end is around 0.87. Since agents can't interact with agents in other sub-simulations (i.e., infected agents in one sub-simulation do not have to check their distance to susceptible agents from other sub-simulations), greater numbers of sub-simulations have a similar effect as spatial binning with respect to scaling. This is why the slopes are around 1 for the CPU code and the end of the GPU code.

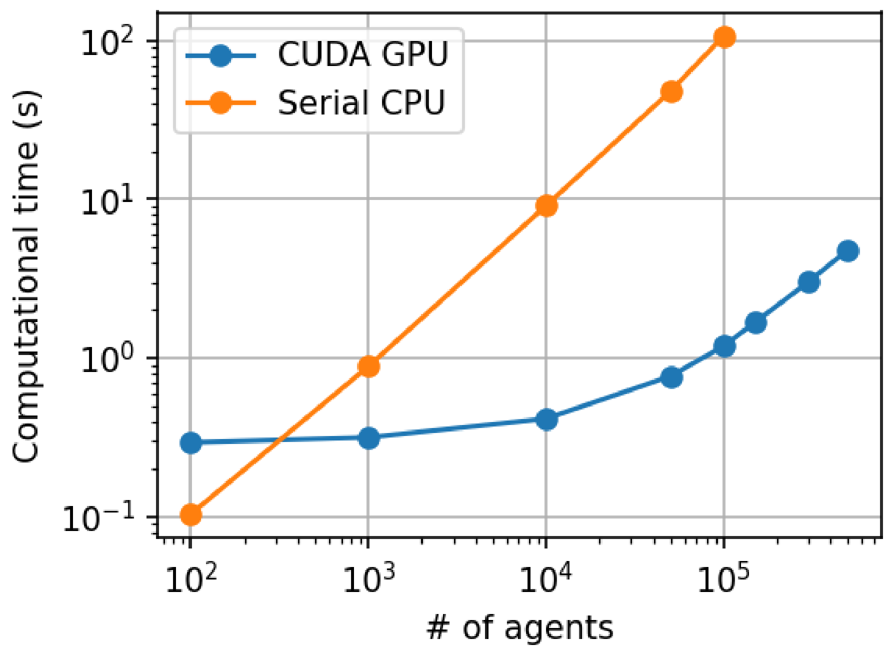


Figure 2: Computational time of GPU code as the number of agents *n* increases proportional to the number of sub-simulations (100 agents/sub-simulation).

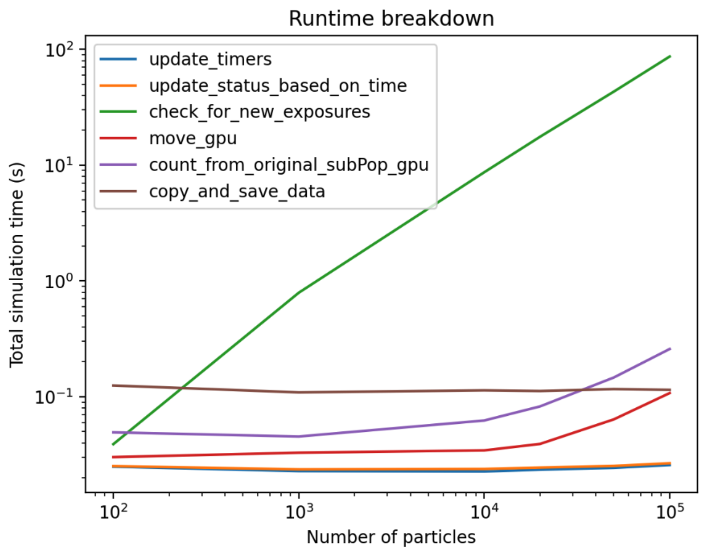
Runtime breakdown

In this section, we analyze the runtime of different components of the CUDA GPU parallelized code. The runtime breakdown is provided in Figure 3, which consists of two subfigures: Figure 3a represents the runtime breakdown with a single sub-simulation, while Figure 3b shows the breakdown with multi-sub-simulations.

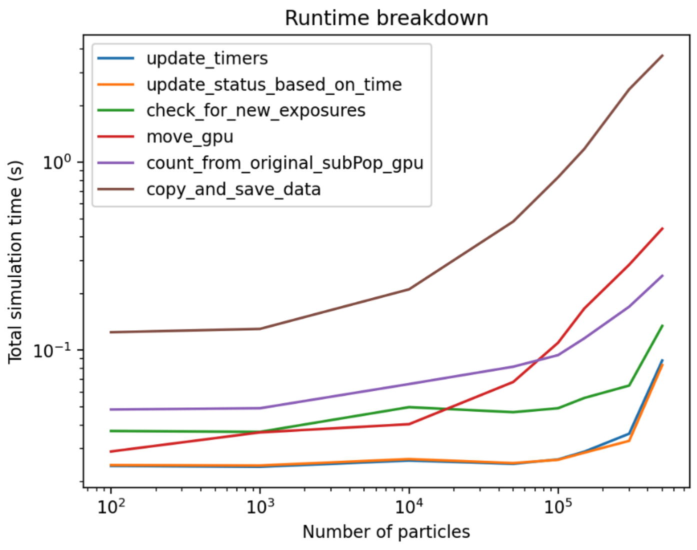
For the single sub-simulation case (Figure 3a), the majority of the runtime is dominated by the `check_for_new_exposures` function, accounting for a significant portion of the total runtime. This is expected, as the kernel is responsible for checking the distances between susceptible and infected agents, which requires a considerable amount of computation. As the number of agents increases, the time spent on this function grows as well, further contributing to the overall runtime.

For the multi-simulation case (Figure 3b), the runtime breakdown exhibits a starkly different pattern, with the `copy_and_save_data` function taking up the majority of the runtime. We note that in the multi-simulation framework, arithmetically intensive kernels such as `move_gpu` take up a larger percentage of the total runtime compared to the single simulation case. Kernels consisting mostly of conditional statements like `check_for_new_exposures` take up a much smaller percentage of the total runtime, partly due to the fact that agents in different sub-simulations cannot interact with each other, reducing the number of distance checks needed.

Overall, the runtime breakdown analysis suggests that the bottlenecks in the simulation vary and depend on the type of simulation being executed. This highlights the importance of optimizing different components of the code to achieve better performance for both single and multi-simulation scenarios.



(a) Runtime breakdown **with a single sub-simulation**.



(b) Runtime breakdown **with multiple sub-simulations**.

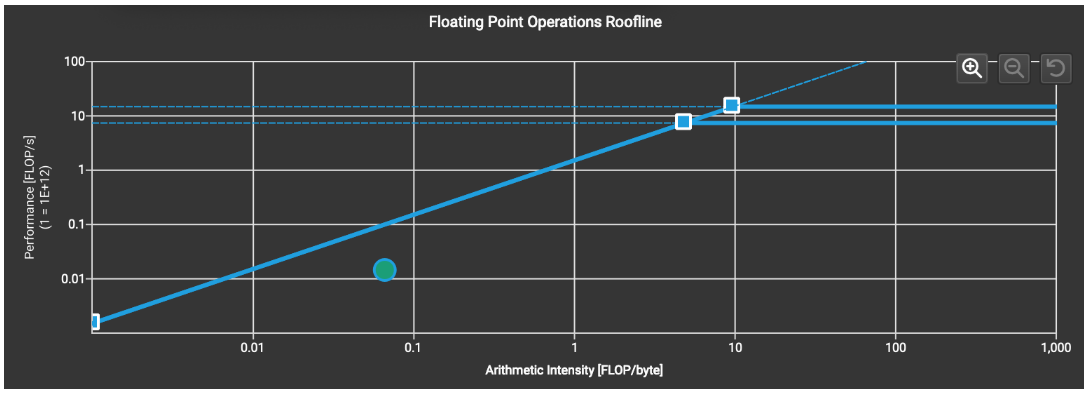
Figure 3: A runtime breakdown on the CUDA GPU parallelized code, describing each of the six sections of the code—`update_timers`, `update_status_based_on_time`, `check_for_new_exposures`, `move_gpu`, `count_from_original_subPop_gpu`, & `copy_and_save_data`—as the number of particles is increased.

Roofline analysis

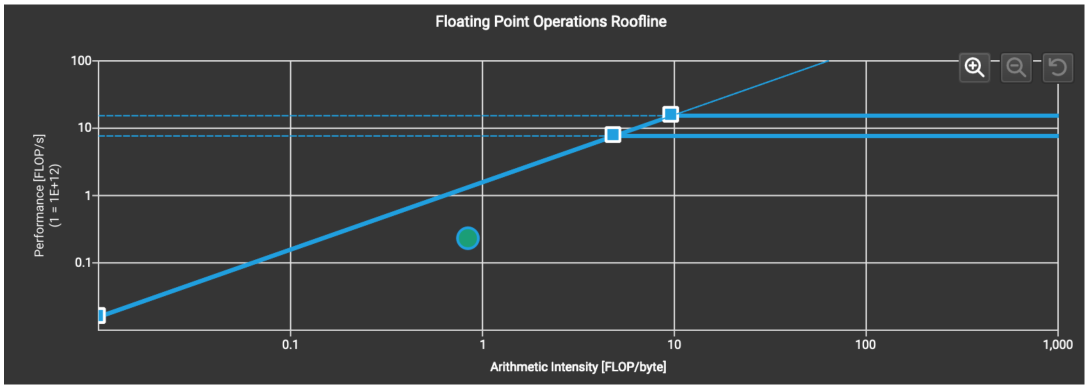
We ran a roofline analysis using NVIDIA Nsight Compute. This profiler significantly slows down code, so it was not feasible to run an entire simulation with the profiler. To get a sense of performance across different parts of the simulation, we ran three versions of inputs. In the first (EARLY), representing the start of an outbreak, the initial disease status fractions are: S = 0.99, E = 0, I = 0.01, and R = 0. In the second (MIDDLE), representing the middle of an outbreak, the initial disease status fractions are: S = 0.5, E = 0.25, I = 0.25, and R = 0. In the third (LATE), representing the end of an outbreak, the initial disease status fractions are: S = 0, E = 0, I = 0, and R = 1. We ran these with 500,000 agents split evenly across 5,000 sub-simulations, each containing a single sub-population. Jumping probability was 0 (as there was only one sub-population in each sub-simulation), mobility was 0.1, and random seeds were assigned 0-5000.

The NVIDIA NSight Compute automatically runs a roofline analysis on each CUDA kernel individually. Some of the kernels' arithmetic intensity is too low to show up on the roofline plot (specifically, `count_from_original_subPop` and `update_status_based_on_time`).

Of the other kernels, we first examine the `check_for_new_exposures` kernel (Figure 4), which is responsible for checking if any susceptible agents are within the infection radius of infected agents, and if so, changing their status to exposed. We see that the double precision achieved value of the kernel, in both the early and middle scenarios falls within the memory-bound region of the plot (plot not available for late scenario), indicating that if we wanted to further utilize the computational capacity of the GPU,



(a) **EARLY**: Roofline analysis of `check_for_new_exposures` CUDA kernel.



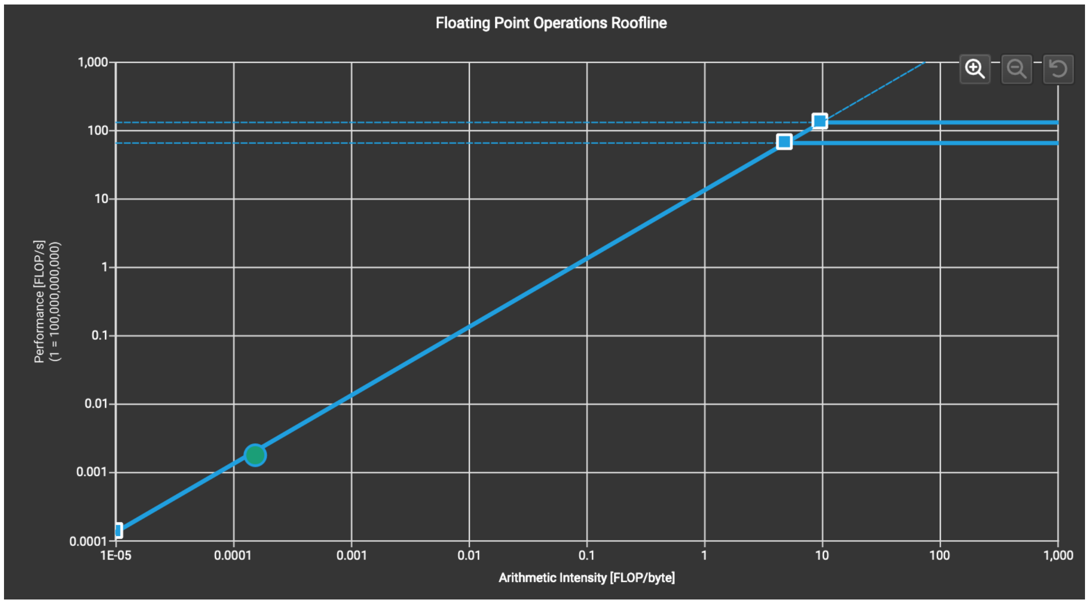
(b) **MIDDLE**: Roofline analysis of `check_for_new_exposures` CUDA kernel.

Figure 4: Roofline analysis of `check_for_new_exposures` CUDA kernel for different stages (green dots: double precision achieved value).

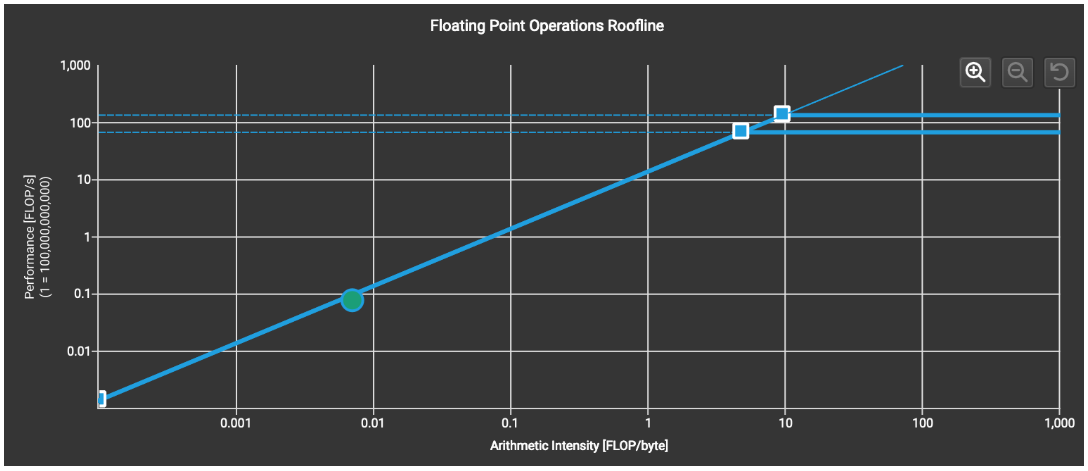
we would need to improve the memory efficiency of this kernel. There is also still room to improve the performance (FLOP/s) without increasing the arithmetic intensity, as the current performance in both the early and middle cases fall below the memory bandwidth boundary. The arithmetic intensity is higher in the middle case than the early case, which makes sense, as the number of computations increases with higher numbers of infected and susceptible agents. This indicates that we would be able to increase the arithmetic intensity of this kernel by only running on infected agents (as mentioned previously in the Opportunities for improvement section), rather than running the kernel over all agents and returning if they are not infected.

Next, we look at the roofline analysis of the `update_timers` kernel (Figure 5), which increases the timers of exposed and infected agents by Δt. We again observe that the performance is memory-bound in both the early and middle scenarios. However, the performance in this kernel is much closer to the memory bandwidth boundary than in the `check_for_new_exposures` kernel. Therefore, if performance was to be noticeably improved, the arithmetic intensity would first need to be increased. Again, the arithmetic intensity is higher in the middle case than the early case, since the timers only need to be updated for infected and exposed agents.

Lastly, we will look at the roofline analysis of the `move_gpu` kernel, which is responsible for moving the agents each time step (Figure 6). The roofline analysis looks about the same for each scenario, since all agents move at every time step, and the behavior is not time or status dependent. Again, the performance is memory-bound. There is room for the performance to be improved without increasing the arithmetic intensity.

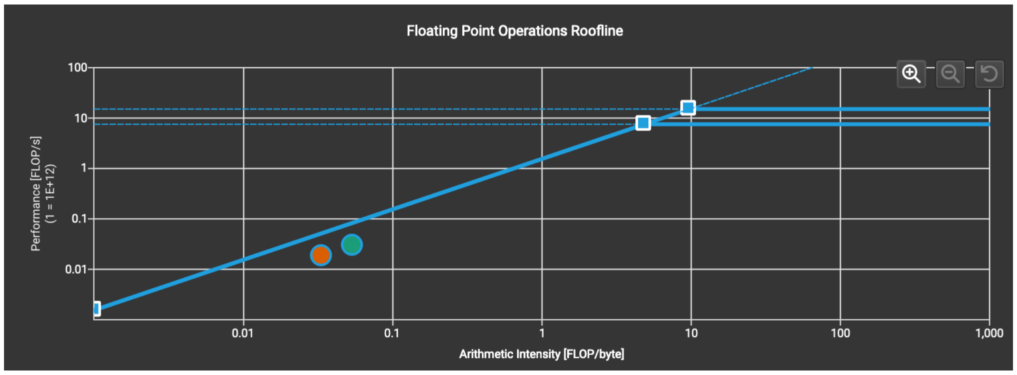


(a) EARLY: Roofline analysis of `update_timers` CUDA kernel.

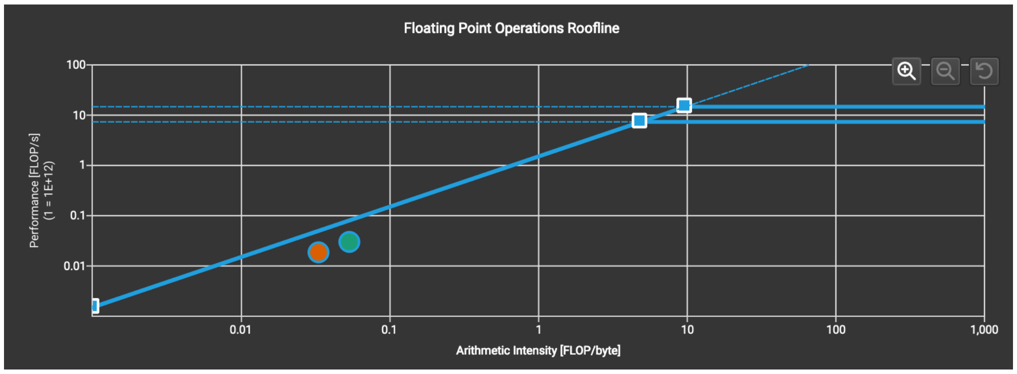


(b) MIDDLE: Roofline analysis of `update_timers` CUDA kernel.

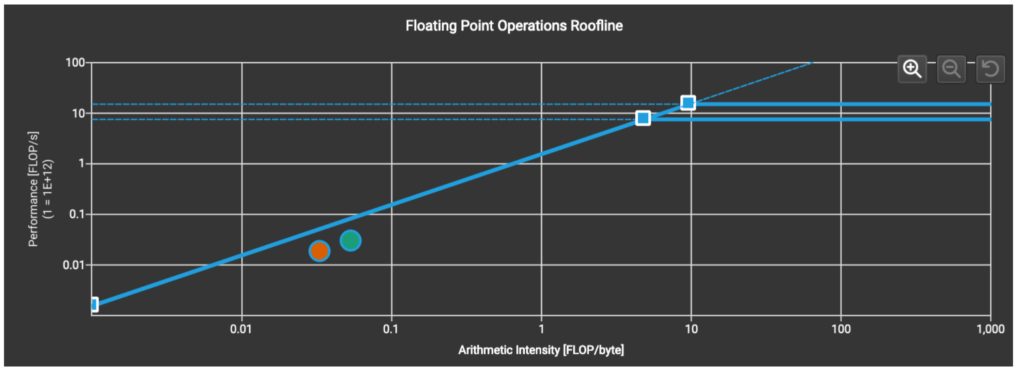
Figure 5: Roofline analysis of `update_timers` CUDA kernel at different stages of execution (green dot: double precision achieved value).



(a) EARLY: Roofline analysis of `move_gpu` CUDA kernel.



(b) MIDDLE: Roofline analysis of `move_gpu` CUDA kernel.



(c) LATE: Roofline analysis of `move_gpu` CUDA kernel.

Figure 6: Roofline analysis of `move_gpu` CUDA kernel at different stages of execution (red dot: double precision achieved value, green dot: single precision achieved value).

Discussion & Conclusion

Our results demonstrated the computational efficiency and scalability of ABM disease spread simulations using parallelized CUDA programming, as compared to the serial CPU implementation. At smaller population sizes, our results indicate that the serial implementation is faster than the parallelized implementation, likely due the higher single-core performance of a CPU relative to a GPU. However, as the population sizes are scaled, the GPU code quickly surpassed the CPU. These results provide insights on the critical number of agents within a simulation that may justify the use case for either hardware.

In conclusion, we implemented a serial and parallel agent-based modeling simulation for the modeling and analysis of disease spread. We can use these implementations to rapidly simulate a large number scenarios that can enhance our understanding of disease transmission and inform effective mitigation efforts.

Appendix A: More details on the data structures

The `multisim` code performs a number of simulations each with sub-populations. To begin, the command line parameters include both the input and output file. A full list of the command line arguments is in Table 2.

Argument	Type	Description	Default/Example Input
`-o`	`string`	Output file name	"`save.out`"
`-file`	`string`	File path to population info csv	"`/path/to/multisim.csv`"
`-T_E`	`double`	Mean exposure time	`11.6`
`-T_E_stdev`	`double`	Standard deviation of exposure time	`1.9`
`-T_I`	`double`	Mean infection time	`18.49`
`-T_I_stdev`	`double`	Standard deviation of infection time	`3.71`
`-d_IU`	`double`	Infection radius	`0.005`
`-dist`	`string`	Type of distribution	"`Gamma`"
`-T`	`double`	Total simulation time	`300`
`-del_t`	`double`	Time step size	`0.1`
`-save_data_for_rendering`	`boolean`	Save data for rendering	"`True`"

Table 2: Command line arguments

Additionally, there a set of arguments stored in the input csv file (e.g., `-file`). These arguments include information about the jumping probabilities between sub-populations and mobility parameters. A full list of these arguments is in Table 3.

Argument	Type	Description	Example
Total population	`int`	Total number of agents within a sub-population	100
`S`	`double`	Percentage of **Total population** that is *susceptible*	0.99
`E`	`double`	Percentage of **Total population** that is *exposed*	0.99
`I`	`double`	Percentage of **Total population** that is *infected*	0.99
`R`	`double`	Percentage of **Total population** that is *recovered*	0.99
`X_neg_lim`	`double`	Lower x bound of agents	0
`X_pos_lim`	`double`	Upper x bound of agents	0.1
`Y_neg_lim`	`double`	Lower y bound of agents	0
`Y_pos_lim`	`double`	Upper y bound of agents	0.1
Sub-simulation	`int`	Sub-simulation ID	2
Seed	`int`	Random seed for the sub-simulation	99
M	`int`	Total number of sub-populations for that particular sub-simulation	2
Mobility	`double`	Length of vector that agents jump in a second	0.01
Jumping prob	`double`	Probability that an agent jumps to a new sub-population	0.25

Table 3: Arguments within input `csv` file

References

[1] Nicholson Collier, et al. “Large-Scale Agent-Based Modeling with Repast HPC: A Case Study in Parallelizing an Agent-Based Model”. Ed. by Sascha Hunold et al., Springer International Publishing, 2015, pp. 454–65.

[2] Jonathan Ozik, et al. “A population data-driven workflow for COVID-19 modeling and learning”. *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, 2021, pp. 483–99. *https://doi.org/10.1177/10943420211035164*, https://doi.org/10.1177/10943420211035164.

[3] Paul Richmond and Mozhgan K. Chimeh. “FLAME GPU: Complex System Simulation Framework”. *2017 International Conference on High Performance Computing & Simulation (HPCS)*. 2017, pp. 11–17, https://doi.org/10.1109/HPCS.2017.12.

[4] James Demmel. CS267 Spring 2023 - HW2-3. *Parallelizing a Particle Simulation*, sites.google.com/lbl.gov/cs267-spr2023/hw2-3.