GPU-ACCELERATED INFLUENZA SIMULATIONS FOR OPERATIONAL MODELING

Peter Holvenstot, M.S.

Western Michigan University, 2014

Simulations of influenza spread are useful for decision-making during public-health emergencies. Policy-makers use models to predict disease spread and estimate the effects of various intervention strategies. Effective modeling of targeted intervention strategies requires accurate modeling of individual-level behavior and transmission. However, this greatly increases the computational costs of these agent-based models. In addition, if the models are used as an outbreak progresses, some operational decisions must occur rapidly in order to contain the spread of the disease.

Graphics Processing Units (GPUs) are a type of specialized processor used to drive graphical displays. Many recent devices also allow users to write and execute programs using the GPU's processor. This processor is designed with functional units that perform the same operation on many pieces of data at once, providing significant increases in processing power. However, good performance depends on effective usage of the GPU architecture, and not all algorithms can be accelerated using this approach. To date, most influenza simulations have had very limited gains from GPU acceleration.

This thesis investigates the performance of influenza simulations when accelerated by GPU devices. We implement a simulation using an agent-based model of low computational complexity, as well as a framework designed to support creation, processing, and analysis of simulations. We further investigate several approaches for accelerating the processing of this simulation using commonly-available hardware. We discuss strategies for adapting such models to GPU computation, and demonstrate an implementation of our model running entirely in GPU memory. This accelerated implementation supports simulation of very large populations and produces excellent performance., reaching speedups of approximately 95x. Additionally, we implement an OpenMP parallelization of the baseline model and compare results.

GPU-ACCELERATED INFLUENZA SIMULATIONS FOR OPERATIONAL MODELING

by

Peter Holvenstot

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science
Computer Science
Western Michigan University
August 2014

Thesis Committee:

Elise de Doncker, Ph.D., Chair

John Kapenga, Ph.D

Diana Prieto, Ph.D

ACKNOWLEDGMENTS

I'd like to express my appreciation for my advisor, Prof. Diana Prieto, who has supported me at every step of this project. Additionally, I'd like to thank Professors Elise de Doncker and John Kapenga for their help on this and other related work.

The resources of the Western Michigan University High-Performance Computational Science Laboratory were invaluable in completing this project. The HPCSL was established with support from NSF grant 1126438 and the CUDA Teaching Center Grant from NVIDIA.

Peter Holvenstot

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

There is a need for informative simulations to support health policy decision-making during emergent disease outbreaks. Severe outbreaks may require a policy response from public health officials, and models have proven to be useful to project the effects of proposed policies and support recommendations on appropriate courses of action [14]. Using observation data collected in the field, simulations can calibrate their internal state to reflect the extent of an ongoing outbreak. As the spread of the disease increases, effective response becomes more difficult and models must provide statistically confident results as quickly as possible. Operational decision-making may occur in cycles of 4-6 hours or less.

Prieto et al establish several criteria which make influenza models suitable for operational decision-making [12]. First, they must incorporate real-time field observation into an accurate epidemiological and population model. The assumptions of the model must support accurate decision-making in the desired area, but should also be computationally tractable in order to support rapid processing. Simulation of behavior at the individual level is an important component of this, particularly when testing intervention strategies at the location or individual level. Simulations must also be capable of scaling to support large populations while running sufficiently quickly. In order to maximize accessibility, these simulations, this should be able to run on personal computers within the given time constraints.

Several types of simulation models are available, but none reach an adequate balance for operational decision-making. Diffusion-based models offer rapid processing times by coarsening the granularity of the simulation. These simulations work with groups of people assigned into compartments, rather than modeling individual behavior. However, this reduces the ability of researchers to model fine-grained location or individual intervention strategies. Agent-based simulations model the spread of the disease through interactions between specific individuals in the contact network which allows greater ability to model individual behavior and complex mitigation strategies. However, their memory usage and computational burden often necessitates computation on clusters of computers, which increases the barriers to access these simulations. Due to their computation cost, agent-based models are yet to be adapted to support rapid operational decision making.

Graphics Processing Units (GPUs) are a type of specialized processor used to drive graphical displays. Over time many of these devices have become capable of executing arbitrary programs on the GPU hardware. By performing the same operation on many pieces of data at once, this can provide a significant increase in processing power. However, good performance depends on effective usage of the GPU architecture, and not all algorithms can be accelerated effectively using this approach.

In this thesis, we present a high-performance framework for simulating influenza outbreaks. We first review the literature on existing frameworks for accelerating influenza simulations and describe the baseline simulation model used in our research. We also present baseline single-threaded and multi-threaded implementations using CPU-based parallel processing. We then introduce our implementation of the simulation using GPU computation, discuss strategies used, and compare performance to CPU implementations.

2 Literature Review

Models can be generally divided into population-based and agent-based models.

In the population approach, the simulation does not extend down to the granularity of individuals. Instead, cities or other large sub-populations are used as compartments, and disease spread is modeled by equations describing population transitions between compartments. This approach produces results very quickly, and can be effective at high-level forecasts. Available models include the Global Epidemic Model of [6] and the Spatiotemporal Epidemiological Modeler of [5]. [3] published GLEaMviz, which incorporates real-word population and mobility data with a framework to assist in creating and processing simulation configurations. However, such models are of limited utility for operational modeling as they do not simulate individual behavior and cannot implement fine-grained intervention strategies.

The other approach focuses on modeling the behavior of individual people, referred to as agents. In this type of simulation, the spread of disease is controlled by contact networks. Usually, agents follow an independently-generated schedule which specifies their location. An individualized contact network is generated based on other agents present at visited locations. This network is then used by the contact model to stochastically spread disease between agents. This type of model is ideal for operational modeling, because interventions can be used to adjust the contact networks and disease transmission probabilities for individuals.

The contact network and the epidemiological model are important factors in the runtime as well as the epidemiological accuracy of the simulation. Existing work such as EpiSimdemics uses a contact model with relatively high complexity. In this model, a susceptible individual has a chance of becoming infected each time they visit a location. The probability of infection depends on many factors, such as the viral shedding of the individuals present, the susceptibility of the victim, total amount of contact with infectious individuals present, and location-specific constants representing closeness. This requires accurately computing the entire contact network for each susceptible individual during each visit, which increases computational complexity. Additionally, the accuracy depends on epidemiological constants used in the calculation. When fitting these models to real-world data, it may be necessary to perform optimization to fit these constants to observed data.

As the existing work is of a high computational complexity, agent-based simulations such as EpiSimdemics usually must be processed in a distributed-memory cluster processing environment due to memory and time requirements. In this strategy, the simulation is divided into a group of sub-programs called nodes, which are often distributed across multiple computers within a cluster environment. Each node can access only its own memory, and program control signals and data must be explicitly sent between nodes. Tools based on the Message Passing Interface specification are often used to implement this. The advantage of this is that a vastly greater amount of computing resources can be brought to bear on a problem. Programs may utilize the memory and processors of hundreds or thousands of clustered computers. The downside is a much greater workload for the user, who must explicitly give a suitable partitioning and communications scheme for the problem. Also, access to a cluster environment can be expensive or simply unavailable to researchers. This approach is used in much of the published work, including EpiSimdemics as well as several other generalized frameworks for agent-based model processing such as ABM++[13].

Several works have been published using GPU acceleration in distributed-memory processing. A GPU-accelerated version of EpiSimdemics was published in 2012 but achieved speedups of only 3.3x[2]. Additionally, a distributed-memory simulation was published by Zou et al[16]. This simulation produced speedups of 7.4x-11.7x over CPU implementations on simulations of 20 million individuals. However, such simulations still require access to a cluster computing environment.

3 Model

Our simulation is an agent-based model of co-circulating influenza strains within a human population. A detailed description and performance of the model has been documented in Prieto et al[11]. In this section, we summarize the most relevant features.

In the model, agents transition independently between Susceptible, Infected, and Recovered states. At initialization, all agents begin in the susceptible status. A small random initial population of agents is selected to initiate the outbreak. Agents with an infected status make contacts with

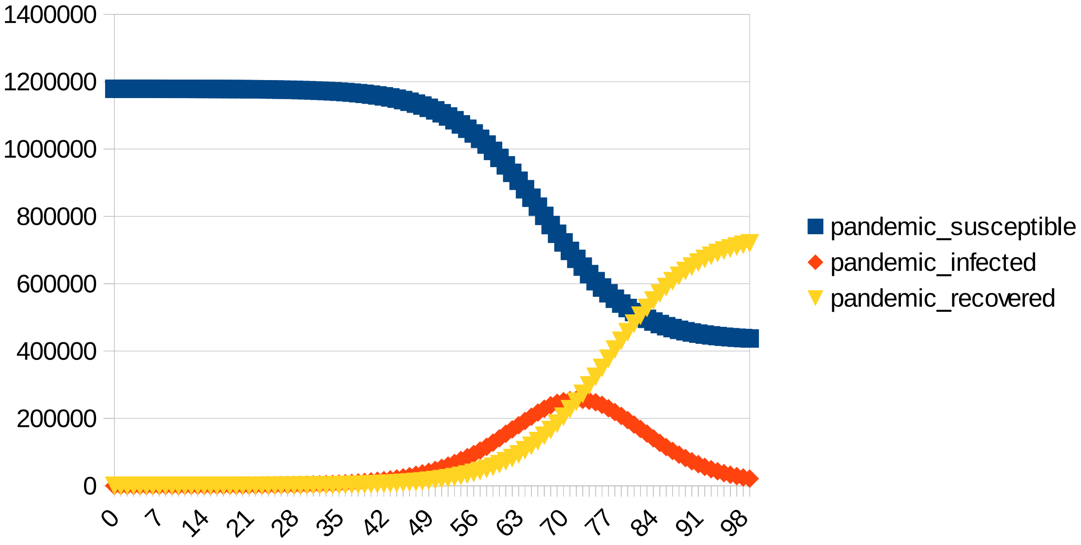

Figure 1: Agent Status During Simulation

other agents, which represent possible opportunities for the disease to spread. After a culmination period of 10 days, agents transition to a recovered status.This status data is maintained separately for both pandemic and seasonal strains of the virus.

Each agent is assigned a schedule which specifies its location for each hour. At certain hours of the day, agents with an active infection will randomly select other agents at their present location as possible infection contacts. At the end of each day, these contacts are processed and infections may occur.

3.1 Agents

The simulation generates a synthetic population based on the demographics of Hillsborough County in Tampa, Florida, information from the 2002 US Economic Census, the 2001 American Community Survey, and the 2001 National Household Travel Survey. The population size is given by the input number of households. Each household is assigned a number of adult and children members. Children are further assigned an age group and an age-appropriate school, while adults are assigned a primary workplace.

In order to simulate disease transmission, realistic contact networks that accurately reflect interpersonal contact are needed. In agent-based models, these contact networks are generated from the individuals present at the agent's location. A scheduling algorithm determines the locations agents will visit throughout the day, and this is used to implement the contact model.

Our scheduling algorithm was designed to mimic a normal work/school attendance pattern. On weekdays, adults will attend their workplace from 8am to 5pm. At 6pm and 7pm, the adult will run errands to 2 randomly-selected destinations. Children will attend their school from 8am to 5pm, and will attend one randomly selected afterschool activity for two hours from 6pm to 7pm. On weekends, each agent will run errands to 3 different destinations during one of 10 possible hours. Otherwise, agents will remain at home.

Schedules are nominally determined on an hourly basis, but in order to improve performance locations are only generated for the hours during which contacts are made. During weekdays, 4 time-periods are simulated, representing work, errand1, errand2, and home. During weekends, 11 time periods are simulated, representing the 10 possible errand hours, and home contact.

3.2 Epidemiological Model

For each strain, a basic reproduction number is given as an input parameter. This represents the average number of further infections that a person will generate throughout their period of infectiousness, assuming that the entire population is susceptible. Another parameter given is viral shedding, which is used as a proxy for infectiousness. The amount of viral shedding occurring on each day between infection and recovery is expressed as a fraction of the total viral shedding throughout this period. This value is used to distribute the expected number of infections given by the basic reproduction number throughout the period of infectiousness, yielding an expected number of infections for each day. The expected daily reproduction number is further distributed through the daily contacts of an infected case. This yields a probability of successful transmission to each contact.

After the simulation has run for a specified number of days, a reproduction number is calculated for each generation of each strain. The reproduction number $\widehat{R}_o$ for generation k is calculated by the ratio between the number of infected cases in the generation $k+1$ and the number of infected cases in a generation k. We conclude that the model is calibrated when the basic reproduction number introduced to the simulation is statistically similar to the maximum value of $\widehat{R}_o$ obtained across all generations. This calibration approach has been further tested and explored in [11].

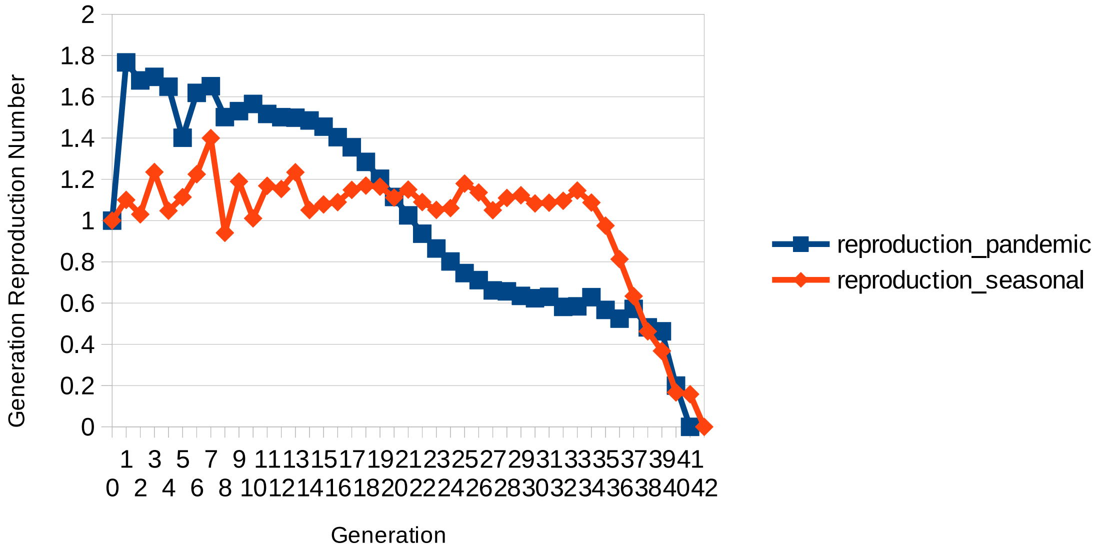

Figure 2: Generational Reproduction Numbers

3.2.1 Parameter Calibration

Several features distinguish this epidemiological model from past approaches. In previous approaches, rate of infection cannot be directly controlled, but instead is a second-order result of epidemiological parameters. For each location visited by a susceptible agent, they will have a probability of becoming infected which is calculated based on several factors, such as agent susceptibility, duration of exposure to viral shedding, and location-specific factors. In order to adjust disease reproduction to match observed field data, these parameters must be calibrated using an optimization process. This optimization process often increases model complexity and computational expense.

To reduce model complexity, our model views transmission from the perspective of infected individuals. Rather than modeling the exposure of susceptible individuals to the virus, we model transmission from infectious individuals to specific individuals within their contact networks. During an influenza outbreak, the infected population roughly follows an exponential growth pattern, as each infected individual infects a number of additional individuals. The average of infected individuals per case called the basic reproduction number, and is available from field observational data. Our model uses this observed reproduction number to directly assign a probability of successful transition to each contact within our network.

Infectiousness of an individual will vary throughout the course of the infection. To model this, we use viral shedding as a representation of the relative infectiousness of an individual. Carrat et al provides measurements of the viral shedding for each day of the infection period[4]. By summing these to find the total viral load, we can calculate the fraction of total viral shedding which occurs on each day between infection and recovery. This value is used to distribute the calibration reproduction number across the period of infectiousness, yielding an expected daily reproduction value for an individual. For example, suppose three days have elapsed since an individual's infection. If the shedding profile assigns 5%, 10%, and 25% of total shedding to days 1, 2, and 3 and a value of 2.0 is used as a calibration reproduction number, then this individual would be expected to cause 0.1, 0.2, and 0.5 infections on day 1, 2, and 3 respectively.

The expected daily reproduction value is used to assign a probability of transmission to each contact within the infector's contact network, considering that the number of infected cases follow a binomial distribution, and each susceptible contact is equally probable to result in infection. In a simplified example, assume the above individual has 5 contacts in their network The expected daily reproduction of 0.5 gives each contact a 10% chance of success. In order to represent contacts with differing probabilities of success, each contact is assigned a kappa value which reduces the probability of infection based on the type of activity the contact represents. This calibration approach has been further explored in Prieto et al[11].

3.2.2 Contact Network

At each location they visit, infected agents will randomly select contacts from the location's population as possible disease transmission opportunities. The number of contacts selected is determined by the activity the agent is performing at a location. These are estimated based on a survey of daily contact patterns from European countries by [9], which states that 23% of contacts are made at home, 21% at work, 14% at school, and 16% during leisure activities. Additionally, it gives the mean number of contacts as ranging between 7.95 and 19.77 depending on country, with weekdays having 30% to 40% more contacts than weekends. Since such data is not available for American social contacts, we assume that the results are similar.

A set of contacts based on the survey of Mossong et al[9] was calculated in Prieto et al[11] During a weekday, each adult will select 3 contacts at their workplace, 2 contacts while running errands, and 3 contacts from their household. Each child will select 2 contacts at their school, 2 contacts at their afterschool activity, and 3 contacts from their household. During a weekend, both children and adults will select 2 contacts on their errands and 3 contacts from their household.

These contacts are selected randomly from individuals who are present at their locations without regard to status of the selected individuals. Agents may not select themselves as a contact, and if this occurs the next sequential contact at a location is used instead. It logically follows that no contacts may be selected at a location with a population count of 1, as the agent could not select a contact other than themselves. If this occurs, a null contact is stored and the kappa value is set to zero.

4 Implementations

4.1 Single-Threaded CPU Engine

A single-threaded CPU implementation was developed as a baseline reference, and used for the purpose of model validation as well as performance comparisons to the parallel implementations.

This implementation follows simple object-oriented design principles. The primary object types are Person and Location, which are implemented as classes. Each Person instance contains individual demographic, epistemological, and location data. Location objects are implemented using vector objects to store the people present at each location, and access static arrays which contain information such as the maximum number of contacts that can be made at a location.

The pseudocode for the basic algorithm is given in figure 3. Note this figure is used to represent both the single-threaded and multi-threaded versions. In the single-threaded version, the loops marked as parallel are executed by only one thread. More detailed pseudocode for this algorithm is not given, but is similar to figures 5 and 6 below.

The update loop is executed for all simulated individuals on a daily basis. Individuals whose infection has reached culmination are transitioned to recovered status, and infected individuals are stored for later processing by the contact loop. The agent is then assigned a schedule of 24 locations, which gives their location during each hour. To generate the location objects, these schedule values are used to insert a pointer to the Person object into the proper Location object for each hour.

The other main loop is the contacts loop, which is executed for all infected individuals. This handles the tasks of contact selection, self-calibration of infection probabilities, and determining the success of transmission.

We selected an implementation of the Mersenne Twister Pseudo-Random Number Generator (PRNG) from the Boost.Random library. This algorithm is one of the most popular PRNGs in use today, and provides a good balance of speed and output quality. The PRNG is seeded based on an input file which is different for each replicate in order to provide stochastic behavior for the simulation.

```
setupSim()
for day←0 to max do
  for parallel myIdx←0 to numberPeople do
    updateStatus(myIdx)
    generateSchedule(myIdx)
    insertToLocations(myIdx)
    if isInfected(myIdx) then
      infectedList.insert(myIdx)
    end if
  end for
  for parallel i←0 to numberInfected do
    myIdx←infectedList[i]
    makeContacts(myIdx)
    processContacts(myIdx)
  end for
end for
calculateReproduction()
```

Figure 3: CPU Main Algorithm

4.2 Multi-Threaded CPU Engine

OpenMP is a library which supports shared-memory parallel programming in C++. Compiler #pragma attributes are used to mark certain portions of the program as parallel. When the program reaches these sections, the main program launches a specified number of threads and waits until they have all terminated. The OpenMP thread group then enters the parallel section and distributes the processing of data items between themselves. Using OpenMP to parallelize a program is a straightforward task in theory, but presents some common problems.

First, programmers must ensure their program is thread-safe, meaning that using multiple threads will not cause data to be manipulated in unsafe ways. A common example is a race condition, in which multiple threads attempt to update a value in shared memory at the same time. OpenMP provides several programming approaches to avoid this. Code marked as ”critical” can only be executed by a single thread at a time. Use of mutual-exclusion locks allows multiple threads to enter the critical section, with the programmer guaranteeing that only the thread holding a particular lock may update a piece of data. Most of the C++ Standard Template Library (STL) is not thread-safe, and these techniques must be used when multiple threads are operating on shared STL containers.

The pseudocode for this implementation is also given by figure 3. The scheduling/location generation and contact selection/processing phase were parallelized using OpenMP constructs, and are marked with **for parallel** clauses in the pseudocode. Mutual exclusion was used to prevent thread-safety issues when inserting people into locations and processing contacts. Each thread uses an independent instance of a Mersenne Twister PRNG from the Boost.Random library.

4.3 GPU Engine

GPUs are specialized processors which process data in a highly parallel fashion. While traditional CPUs perform a single instruction on a single piece of data at a time, GPUs perform a single instruction across many pieces of data at the same time. A single CPU might have 4-8 cores per CPU, GPUs may have up to 30 multiprocessor units, each controlling up to 192 cores. While CPU cores usually work independently, GPU cores are dispatched and executed in groups.

There are several commonly available frameworks for executing general-purpose programs on these devices. NVIDIA provides the Compute Unified Device Architecture (CUDA) for their GPU devices, which supports programs based on the C and Fortran programming languages. There is also a standard called OpenCL which is promoted by AMD, but this hasn't been as widely accepted. For our GPU implementation, we chose the CUDA framework.

In the CUDA model, the CPU and GPU each have independent program code and memory spaces. The term host is used to refer to memory and code which can be run on the CPU, and device refers to memory and code which is run by the GPU. Device code is structured into programs called kernels which are run on the GPU processor. In a typical programming pattern, the host will copy some data into device memory, launch one or more kernels to perform computation, and then copy the results back to host memory.

Cores are assigned to computational groups called blocks, which can share on-chip memory to perform tasks cooperatively. Within a block, threads are controlled in smaller groups called warps that execute instructions and memory accesses in lockstep. Many blocks can be resident on a multiprocessor at once, and are dispatched for execution as their data becomes available. These blocks are used in a larger group called a grid. Blocks and grids may be 1D, 2D, or three-dimensional and these dimensions may be controlled by the user.

4.3.1 Design Considerations

The most basic challenges of implementing GPU acceleration arise due to the extremely parallel nature of the processor. Candidate algorithms must exhibit a high degree of parallelism to fully occupy the processor, and simulations which do not are fundamentally unsuited to these devices. Additionally, code branching and thread divergence can substantially reduce processing power, and code should be structured so that all threads follow the same code path as much as possible. Synchronization primitives and atomic operations are relatively limited and improper usage can significantly reduce performance at high degrees of parallelism.

Thread blocks are the most important tool for approaching these limitations. Blocks allow a group of threads to share working memory and use synchronization signals to collectively accomplish tasks. In many cases, tasks approached with synchronization operations in CPU environments may be restructured to use collective operations such as reductions, sorting, or prefix scans which allow high degrees of parallelism. They also provide a means to stage data or results when reading or writing global memory, and allow a drastic reduction in the usage of atomic operations. However, blocks also have certain limitations. Launching kernels from within kernels is called dynamic parallelism, and most devices do not support this capability. Also, inter-block communication is not possible because the GPU scheduler does not provide any guarantees regarding the processing of blocks.

Memory usage is the most significant constraint to scaling the population size on GPUs. CPUs are frequently equipped with very large amounts of memory, on the order of 128GB to 512GB or more. However, GPUs have a very limited amount of onboard memory. Most commodity desktop GPUs are equipped with only 1-2GB of memory, and even very expensive GPGPU compute cards only have 5-12GB. A variety of other factors make it difficult to reach full memory utilization. If the GPU is used as a display device, resources allocated to this task cannot also service the program. Memory must be contiguous blocks, and fragmentation can result from external sources or poor memory usage within the program. Error-corrected memory is available on Tesla compute cards, but causes a 10% overhead to store the correction codes. Many parallel implementations of sort algorithms require additional auxiliary global memory as working space. Additionally, memory bandwidth is a critical limit in algorithm performance. Thus, making efficient use of memory is paramount in order to scale the simulation to large levels.

The most effective strategy to approaching these problems is using a ”flat” memory strategy that uses large chunks of contiguous memory allocated at the start of the program. By allocating sufficient memory to accommodate the program’s peak memory needs, fragmentation is minimized. Additionally, programmers should consider what data actually needs to be persisted using global memory. Storing intermediate data (such as contacts) can consume a great deal of global memory and significantly decreases performance. The use of on-chip thread-local or shared memory relieves pressure on global memory and increases performance by removing read-write cycles.

4.3.2 Implementation

In order to maximize simulation performance, the program was implemented entirely as a GPU-side program. During normal execution, all steps of the simulation are executed by the GPU processor. This approach and implementation details were published in Holvenstot et al[8]. For validation purposes, intermediate results may be stored and checked by the host computer, but this drastically reduces the simulation's performance and supported simulation size.

The algorithm of 4 was implemented using a mixture of kernels and device-wide collective operations. Individual kernels handled status updates, scheduling, and contacts. The Thrust template library was used to implement location generation functionality, including sorting and vectorized searching, as well as a selection operation to create the list of infected agents. Optionally, the CUDA Un-Bound library provides a small increase in sorting performance, but requires additional memory which reduces the supportable simulation size.

Where appropriate, simulation data is stored in device constant memory. This is a fast in-processor data cache that performs well when all threads in a warp attempt to access the same memory location. This is useful for data such as probability density functions.

4.3.3 Scheduling and Location Generation

Assigning schedules and generating locations is a dominant portion of program runtime, so this is a critical portion where good performance must be achieved. In the CPU implementation, locations are generated by inserting pointers to people into the location's vector. For the OpenMP implementation, this was parallelized using mutual-exclusion locks to ensure thread safety. However, this approach is unsuitable for the GPU synchronization and memory models.

Instead, this task is broken into several parts which can be parallelized effectively. First, schedules are generated and copied unsorted into global memory. Next, a device-wide sort algorithm sorts the personID numbers using the locations as a key. Each location is now represented as a contiguous region of personIDs somewhere within a large array. Finally, a vectorized lower-bound search is used to identify the array index of the first person scheduled at a given location number, which is referred to as the location offset. The offset of location l and the offset of location $l + 1$ together define the bounds of the region within the array.

In order to avoid the use of global memory to store selected contacts, all locations for an entire day must be available at once for the contact selection kernel. Thus, we must represent both location and time when generating the location arrays. In the initial implementation, this was represented as a tuple $(hour, locationID)$, but by encoding these together we realized a substantial improvement in performance. A scheduleID value can be calculated as $(hour * numberLocations) + locationID$, and represents one location at one hour. This value can be used directly by any sort-by-key algorithm and provides much greater performance than a custom comparator.

4.3.4 Pseudo-Random Number Generator

As a stochastic simulation, this program relies on a Pseudo-Random Number Generator (PRNG) in order to generate random numbers used for scheduling, contact selection, and determining the success of transmission attempts. We elected to use a Counter-based Pseudo-Random Number Generator (CBPRNG) due to several advantages they offer, and we selected the Random123 library for our implementation[15].

Stateful PRNGs generate outputs from an internal state which is stored and permuted after each output generated. Instead, Counter-based Pseudo-Random Number Generators (CBPRNGs) use encryption algorithms to generate outputs. Rather than relying on internal state, a key is used to encrypt a counter value into a random number, and each counter value produces a different output. To put CBPRNGs into the terminology of stateful PRNGs, the key is the seed, and the counter value represents how many times the PRNG has been called. However, rather than requiring N state permutations for the Nth call, CBPRNGs always complete in O(1) time.

This gives several advantages to stochastic agent-based simulations. First, the algorithm parallelizes well. Since there is no saved state, each call to the PRNG operates only on the input parameters. There is no need to explicitly generate and store a global or block-specific buffer of random numbers, and there is no overhead from initializing and tearing down PRNG states nor the need to store them in either processor-shared memory during the run or global memory between runs. RNG values are simply generated when entering a stochastic portion of the algorithm. Since the PRNG algorithms are highly computation-intensive, this offers an ideal opportunity to hide memory latency and increase arithmetic intensity in memory-bound portions of the algorithm.

The counter value can be chosen in ways that provide additional advantages. In our simulation, the output value of a PRNG generation is dependent on the position of the data item being processed. If an algorithm requires C random numbers per data item processed, the first counter input value for each item can be expressed as $ctr = globalOffset + (dataIndex * C)$ First, this means that the RNG stream for the program is hardware-independent. Arbitrary numbers of cores can be applied in any compute grid configuration, which allows flexibility in tuning performance across varying hardware. Additionally, the global offset value used to generate a sequence of RNG numbers can be stored and used in later parts of the algorithm. This can be used to re-generate arbitrary sections of the RNG stream with low computational complexity.

```
setupSim()
for day←0 to max do
    countAndRecover()
    assignSchedules()
    generateLocations()
    selectInfected()
    makeAndProcessContacts()
end for
calculateReproduction()
```

Figure 4: GPU Main Algorithm

4.3.5 Pseudocode

Figure 4 gives a high-level overview of the algorithm. In the setup portion, a synthetic population is generated and the initial infection is seeded. The simulation then enters the main loop, which simulates interaction and disease transmission between agents for a specified number of days. Once the simulation has been completed the program performs final calculations, and exits.

The daily loop makes up the main portion of the simulation, and consists of a series of operations which are carried out during each simulated day. One kernel iterates all agents to handle status updates and counting. Agents who have reached culmination are transitioned from infected to recovered status, and the number of agents in each state are counted and used as an output. Schedule items are generated for each agent, and used to construct representations of locations. A contact network will then be generated for each infected agent, and processed to determine the success of transmission.

Since at any given time most agents do not have an active infection, the program must locate these individuals before generating and processing contact networks. This was implemented by using a device-wide selection algorithm from the Thrust library.

Once the selection of infected individuals has been completed, the contact kernel is launched. This handles the task of selecting and processing the contact networks for these individuals. To reduce usage of global memory, these contacts are selected into on-chip memory. This approach substantially increases performance, but does require that all locations throughout the day be available for access.

Figure 5 describes the process of contact selection. Each individual will randomly select contacts from their scheduled locations/hours. Agents may not select themselves as contacts, and if this occurs the next agent at the location is selected instead. An arbitrary algorithm controls the quantity, hours, and kappa value weights of contacts. The algorithm outputs an array of personID values selected as contacts, as well as an array of kappa value weights.

In order for the selection to take place, the schedule of all infected individuals must be available.

```
for i← 0 to contactsDesired do
  myLoc← selectLocationFromSchedule()
  lowerBound← locOffsets[myLoc]
  peopleAtLoc← locOffsets[myLoc+1] - lowerBound
  if peopleAtLoc==1 then
    contactKappa[i]← 0
  else
    personSelected← UnsignedRand() mod peopleAtLoc
    contactTarget[i]← locationPeople[lowerBound+personSelected]
    if contactTarget[i]==myIdx then
      personSelected← (personSelected+1) mod peopleAtLoc
      contactTarget[i]← locationPeople[lowerBound+personSelected]
    end if
    contactKappa[i]← ArbitraryAlgorithm()
  end if
end for
```

Figure 5: Contact Selection Algorithm

```
kappaSum← sum(contactKappa)
baseInfectionThreshold←calibrateInfector(kappaSum,day)
for i← 0 to contactsDesired do
  contactInfectionThreshold←baseInfectionThreshold * contactKappa[i]
  yVal← UnsignedRand() / UNSIGNED_MAX
  if  yVal<contactInfectionThreshold then
    transmitInfection(contactTarget[i])
  end if
end for
```

Figure 6: Contact Processing Algorithm

To save global memory, the re-generation strategy above was used. Using the ID number of a person, the input values used to generate a schedule can be re-constructed and the schedule can be generated into the contacts kernel. This allows us to trade abundant processor power for scarce global memory when performing this step.

Figure 6 gives a simplified pseudocode for processing contacts. Our implementation calculates separate infection thresholds, yVals, and transmission success for both pandemic and seasonal strains of the virus, but we have abbreviated this for readability. In the self-calibration step, the kappa sum, viral shedding profiles, and individual epistemological data are used to calculate the probability of a successful infection for a contact with a kappa value of 1. This base probability is then modified by the kappa value to weight individual contacts.

4.4 Support Framework

A small suite of tools were written to assist in setting up replicates, running them, and analyzing the results.

The most important of these was the batch manager/batch client pair. The batch manager allows the user to specify desired simulation configurations, enqueue a specific number of replicates with different seeds, and extract the output files once computation has completed. The batch client operates on cluster processing nodes, and handles the tasks of getting and processing these replicates, as well as submitting the results once the simulation has completed. These were implemented around a SQL database. The initial implementation was written as a C# program using a SQLite database, but for reasons of platform independence and to support multiple computers a second implementation was written using Python and a MariaDB database.

Additional support tools handle arranging data into desired formats, and processing the data to extract some desired features of the output.

5 Results

The performance of the CPU implementations was measured on a server using dual 8-core 2.6GHz Xeon E5-2670 CPUs in Western Michigan University's High Performance Computational Science Laboratory. The GPU implementation was evaluated on several different devices reflecting a variety of GPGPU-capable hardware. The NVIDIA K20 is a high-end Tesla compute card targeted at high-performance computing. The Nvidia GT640 is a modern but low-end commodity desktop graphics card intended for media PC usage. Finally the Quadro FX 880M is a relatively old chipset used in workstation laptops. The program was built with full compiler optimizations in each configuration. The most important specifications when comparing the performance of these devices is are the number of cores and the memory bandwidth, which are given in Table 1. The final column of the table gives an approximate maximum population scale value which can be supported on the device, as described below.

Table 2 shows the run-time of the single-threaded, parallel GPU, and OpenMP implementations tested across different hardware and simulation configurations. The first two columns show the scale of the simulation. The baseline simulation configuration (scale=1) uses approximately 2.5 million agents (1 million household locations) and 12,800 workplace and errand locations (per hour) with a simulation duration of 100 days. To demonstrate the implementation's performance under other simulation configurations, we allow the population and the number of modeled locations to be independently scaled from this baseline. For example, a population scale of 10 represents 10 million households (approximately 25 million people) and an Location scale of 100 represents 1.28 million locations. Values which have been struck out are the result of a simulation configuration which is too large to fit in the device's memory. The runtime is given as an average computed across 10

Table 1: Device Specifications

Device	Cores	Mem. Capacity	Mem. Bandwidth	Max Pscale
K20	2496	5GB	208GB/s	20
GT640	384	2GB	28.5GB/s	10
Q880M	48	1GB	25.3GB/s	5

Table 2: Runtime Comparison

Simulation Scale		CPU Runtime (sec)					GPU Runtime (sec)		
People	Locations	1-core	2-core	4-core	8-core	16-core	K20	GT640	Q880M
0.1	0.1	4	4.3	3.1	2.6	2.7	0.22	0.88	2.62
0.1	1	4.7	4.7	3	2.3	2.3	0.23	0.9	2.69
1	1	47.7	47	28.9	21.8	20.6	1.25	5.91	24.1
1	10	61.7	51	30.8	23	21.8	1.32	6.25	25.07
5	5	301.5	244.3	144.8	105.9	99.9	5.86	28.62	124.6
5	50	454.3	279.5	161.7	116.2	109.4	6.23	30.13	128.6
10	10	681	506.9	293.5	212	199.9	11.65	57.21	X
10	100	1024.9	589.4	337.2	237.6	221.6	12.34	58.98	X
20	20	1550.5	1070.9	605.8	430.6	403.3	23.08	X	X
20	200	2326.6	1254	715.3	492.5	452.3	24.66	X	X

iterations using different seed values.

Table 3 shows the speedup of the GPU and OpenMP implementations relative to the single-threaded implementation. GPU speedup can be given as kernel speedup, which describes the increase in performance of accelerated portions of the program, and program speedup, which describes the improvement in overall program time from accelerating certain portions. As we have accelerated the entire simulation onboard the GPU, our speedups for the GPU simulation reflect program speedup.

The number of people in the simulation is the dominant factor in the runtime. The CPU can complete small simulations reasonably quickly. However, as the simulation is enlarged its performance begins to degrade slightly. The OpenMP parallelizations display expected results, and a speedup is observed with most configurations. Reasonable efficiency is obtained up to 4 cores, with a small additional speedup from 8 cores, after which the returns diminish.

Within its limits the GPU provides excellent performance and a speedup is observed at all simulation sizes on all devices. The speedup of the GPU implementation increases as the simulation is scaled up until the memory limits of the device are reached. This behavior is likely due to fixed launch/synchronization overhead, which is amortized across larger numbers of elements as the simulation scale increases.

Table 3: Speedup

Simulation Scale		OpenMP Speedup				GPU Speedup		
People	Locations	2-core	4-core	8-core	16-core	K20	GT640	Q880M
0.1	0.1	0.93	1.29	1.53	1.48	18.18	4.54	1.53
0.1	1	1	1.56	2.04	2.04	20.43	5.22	1.74
1	1	1.01	1.65	2.19	2.32	38.16	8.07	1.98
1	10	1.21	2.0	2.68	2.83	46.74	9.87	2.46
5	5	1.23	2.08	2.84	3.01	51.45	10.53	2.42
5	50	1.62	2.81	3.91	4.15	72.92	15.08	3.53
10	10	1.34	2.32	3.21	3.41	58.45	11.90	X
10	100	1.73	3.04	4.31	4.63	83.06	17.38	X
20	20	1.45	2.56	3.6	3.84	67.18	X	X
20	200	1.85	3.25	4.72	5.14	94.35	X	X

6 Future Work

6.1 Interventions

As our work was focused on increasing the accessibility and scalability of models for operational use, we did not implement intervention strategies in our initial simulation. However, we suggest several approaches to incorporate these interventions into the model.

Pharmaceutical Interventions

The World Health Organization describes several strategies that involve the use of vaccination or medication to alter the epidemiological parameters of the model[10].

Antiviral interventions involve the use of medications by symptomatic individuals to reduce their viral shedding loads and delay the peak. As viral shedding is already modeled, these strategies may be implemented by altering by using different viral shedding profiles for treated individuals. As described in section 3.2.2, the model attains calibration to the basic reproduction number $\widehat{R}_o$ when the viral shedding values for each day sum to 1.0. If these values instead sum to a different value ρ, then a case with this profile will instead cause $\rho * \widehat{R}_o$ additional cases.

Vaccination strategies involve treatment of individuals prior to exposure to reduce the chance of contracting influenza. Estimates of vaccine effectiveness are available and often range from 70-90%. The effectiveness may be used to reduce the infection probability threshold when the contact target is vaccinated. For example, if a contact normally has a 10% chance of successful infection and a vaccine is 90% effective, then the contact would have only a 1% chance of causing infection if the target is vaccinated.

Non-Pharmaceutical Interventions

The World Health Organization also describe several strategies to limit exposure to disease or reduce susceptibility to influenza that do not involve medication or vaccination[7].

Social distancing, wearing masks, or disinfection reduce the exposure of compliant individuals to the virus. This can be modeled using a similar approach to vaccination. An estimate of effectiveness in reducing exposure to viral shedding can modify the chance of successful transmission to compliant individuals.

Other strategies involve quarantining, closure of locations, or other modifications to agent scheduling. These require special attention due to the self-calibrating nature of the model. Normally, the infection probabilities for an individual will be calibrated to produce $\widehat{R}_o$ additional infections regardless of the number of contacts within the network. If the number of contacts is reduced, the remaining contacts will simply be assigned a greater chance of successful infection.

In order to alter the target reproduction number, the contact network must be computed without intervention and then compared to the contact network with the intervention. For example, without intervention an adult will make 8 contacts per day, consisting of 3 household, 3 work, and 2 errand. Instead, a quarantined adult will make only 3 household contacts per day. With the assumption that all contacts are equally likely to result in infection, the infection probabilities should be calibrated to produce a reproduction number of ${}^3/_8 * \widehat{R}_o$.

6.2 Scale

The simulation size may be increased somewhat further by reducing the simulation's memory usage. In the location generation step, all schedule items for all individuals are sorted. Since radix sort requires O(N) auxiliary memory, this consumes a large amount of memory at large program sizes. Replacing radix sort with an in-place sorting algorithm would allow the simulation to be scaled further, although this might result in slower performance in a critical portion of the algorithm.

It would also be desirable to increase the size of the simulation much further than incremental gains would allow. A GPU-hybrid approach would allow further scaling using the more abundant host memory while still allowing computation on a single PC. The GPU could be used as a dedicated co-processor for generating schedules and sorting the location arrays. These tasks are compute-intensive and could be accomplished efficiently by the GPU using a relatively small amount of data, then transferred back to the CPU for use in making contacts. Very large simulations could be supported by splitting these tasks into segments capable of fitting into GPU memory and then merging the segments within host memory. Since the GPU parallelization model is generally more restrictive than CPUs, the CPU portions could easily be parallized using an arbitrary number of cores with a multi-threaded approach.

To improve performance at the largest population scales, multiple GPUs could be used in a single machine or within a cluster computing environment. Each device is capable of simulating one or more large population centers within its own memory. With the addition of an algorithm to control travel between these population centers, national or international-scale simulations could be rapidly completed. Low-end devices have an excellent cost-to-memory ratio and the performance remains high, so these devices may be a cost-effective approach to creating large simulations.

6.3 Performance

Since the sort operation used when generating locations consumes a large percentage of the program runtime as well as a great deal of memory, improvements to this part of the program have a large impact on simulation runtime and size limits. GPU parallel sorting is an area of active research and incorporating recent findings will have direct performance benefits in a critical portion of the algorithm. One possible approach would be to pre-process the scheduling data in shared memory immediately after generation. Since the re-generation approach allows the schedule of an individual to be reconstructed, the locations of infected individuals do not need to be explicitly sotred. This has the further implication that there is no longer a need to write the schedule into global memory ordered according to the person's ID. Rather than writing the errand sequence directly into global memory, the errands can be written into shared memory and a collective block operation can be used to pre-sort the errand sequence into a tile. The tile can then be written into global memory, where a merge sort could be used to finish the sort. This removes a complete round-trip to global memory, which may offer performance advantages.

It may also be possible to make effective use of dynamic parallelism and other features of recent GPU devices. For example, it is still necessary to explicitly track which individuals have an active infection. This requires iterating the status arrays each day and consumes global memory. Simply scanning the array and using contact methods on the active individuals is likely not an efficient approach, because even during the peak of an outbreak most individuals are not infected, and during most of the simulation the overall percentage is very low. This leads to a large number of idling threads, which is not efficient. Instead, blocks could cooperatively locate which individuals have active infections and launch additional kernels to process them. This could offer benefits in both performance and memory usage.

7 Conclusions

We demonstrated a model of low computational complexity which incorporates features supporting operational modeling. The contact network incorporates real-world data on interpersonal contact, and is computationally tractable even when many agents are present at a location. The epidemiological model is designed to be easily calibrated using available observational field data, and allows contacts to simulate differing levels of intimacy using a weighting system. Additionally, we provide a framework to enable batches of simulations to be easily dispatched and processed.

Our implementation demonstrates significant improvements in simulation runtime which allow real-time processing. For example, EpiSimdemics reports a processing time of 4021 seconds for a 120-day simulation of 50 million people using 200 processing elements in a cluster computing environment [1]. In contrast, a single computer can execute a 100-day simulation in 1550 seconds using our single-threaded implementation, in as little as 403 seconds using the multi-threaded implementation, or in 23 seconds by using the GPU implementation. This allows simulation models to support policymakers in rapid decision-making cycles during emergent outbreaks.

Our simulation provides a basic framework for implementing scalable, high-performance simulations using GPU parallel computation. We demonstrate speedups of up to 94.4x, which is substantially faster than the 3.3x achieved by Barret et al[2] or the 11.7x by Zou et al[16]. We also achieve large simulation scales of up to 50 million individuals per simulation using a single device. This approach allows large populations to be simulated rapidly.

These implementations also significantly decrease the barriers to performing research using agent-based influenza models. The reduced memory usage and processing time of our model allows researchers to perform simulations using a single PC rather than requiring the use of a cluster-computing environment. Our parallel implementations accelerate processing using multi-core CPUs and GPGPU devices which are commonly present in user PCs. These greatly increase the accessibility of simulations modeling individual behavior.

References

[1] C.L. Barrett, K.R. Bisset, S.G. Eubank, Xizhou Feng, and M.V. Marathe. Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008.

[2] Keith Bisset, Ashwin Aji, Madhav Marathe, and Wu-chun Feng. High-performance biocomputing for simulating the spread of contagion over large contact networks. *BMC Genomics*, 13(Suppl 2):S3, 2012.

[3] Wouter Broeck, Corrado Gioannini, Bruno Goncalves, Marco Quaggiotto, Vittoria Colizza, and Alessandro Vespignani. The gleamviz computational tool, a publicly available software to explore realistic epidemic spreading scenarios at the global scale. *BMC Infectious Diseases*, 11(1):37, 2011.

[4] Fabrice Carrat, Elisabeta Vergu, Neil M. Ferguson, Magali Lemaitre, Simon Cauchemez, Steve Leach, and Alain-Jacques Valleron. Time lines of infection and disease in human influenza: A review of volunteer challenge studies. *American Journal of Epidemiology*, 167(7):775–785, 2008.

[5] Stefan B. Edlund, Matthew A. Davis, and James H. Kaufman. The spatiotemporal epidemiological modeler. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI '10, pages 817–820, New York, NY, USA, 2010. ACM.

[6] Joshua M. Epstein, D. Michael Goedecke, Feng Yu, Robert J. Morris, Diane K. Wagener, and Georgiy V. Bobashev. Controlling pandemic flu: The value of international air travel restrictions. *PLoS ONE*, 2(5):e401, 05 2007.

[7] World Health Organization Writing Group. Nonpharmaceutical interventions for pandemic influenza, national and community measures, January 2006.

[8] Peter Holvenstot, Diana Prieto, and Elise de Doncker. Gpgpu parallelization of self-calibrating agent-based influenza outbreak simulation. In *IEEE High Performance Extreme Computing Conference*, September 2014.

[9] Jol Mossong, Niel Hens, Mark Jit, Philippe Beutels, Kari Auranen, Rafael Mikolajczyk, Marco Massari, Stefania Salmaso, Gianpaolo Scalia Tomba, Jacco Wallinga, Janneke Heijne, Malgorzata Sadkowska-Todys, Magdalena Rosinska, and W. John Edmunds. Social contacts and mixing patterns relevant to the spread of infectious diseases. *PLoS Med*, 5(3):e74, 03 2008.

[10] World Health Organization. Pharmaceutical interventions for rapid containment. http://influenzatraining.org/documents/s15549e/s15549e.pdf.

[11] D. Prieto and T.K. Das. An operational epidemiological model for calibrating agent-based simulations of pandemic influenza outbreaks. *Health Care Management Science*, pages 1–19, 2014.

[12] Diana Prieto, Tapas Das, Alex Savachkin, Andres Uribe, Ricardo Izurieta, and Sharad Malavade. A systematic review to identify areas of enhancements of pandemic simulation models for operational use at provincial and local levels. *BMC Public Health*, 12(1):251, 2012.

[13] Douglas Roberts. ”abm++ software framework”. http://parrot-farm.net/ABM++/.

[14] L. A. Rosenfeld, C. E. Fox, D. Kerr, E. Marziale, A. Cullum, K. Lota, J. Stewart, and M. Z. Thompson. Use of computer modeling for emergency preparedness functions by local and state health officials: a needs assessment. *Journal of Public Health Management and Practice*, 15:96–104, Mar-Apr 2009.

[15] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, pages 16:1–16:12, New York, NY, USA, 2011. ACM.

[16] Peng Zou, Ya-shuai L, Ling-da Wu, Li-li Chen, and Yi-ping Yao. Epidemic simulation of a large-scale social contact network on gpu clusters. *SIMULATION*, 89(10):1154–1172, 2013.

Appendix

Pseudocode

9.1 CPU Algorithms

9.1.1 Main

main()

```
setupSim()
for day←0 to max do
    for parallel myIdx←0 to numberPeople do
        updateStatus(myIdx)
        generateSchedule(myIdx)
        insertToLocations(myIdx)
        if isInfected(myIdx) then
            infectedList.insert(myIdx)
        end if
    end for
    for parallel i←0 to numberInfected do
        myIdx←infectedList[i]
        makeContacts(myIdx)
        processContacts(myIdx)
    end for
end for
calculateReproduction()
```

9.1.2 Setup

setupSim()

High-level method to handle all setup for simulation.

```
readInputs()
setup_generateBusinesses()
setup_generateHouseholds()
setup_generateFixedLocations()
setup_initialInfection
```

setup_generateBusinesses()

Scales number of each type of business according to the location scale, and sets up the location data.

```
numberBusinesses←0
for type←0 to numberBusinessTypes do
    typeCount← baseWorkplaceTypeCounts[type] * locationScale
    workplaceTypeCounts[type]←typeCount
    for i←0 to typeCount do
        wpIdx = i + numberBusinesses
        locationData[wpIdx].locationType← BUSINESS
        locationData[wpIdx].businessType← type
        locationData[wpIdx].maxContacts← workplaceMaxContacts[type]
    end for
    numberBusinesses += typeCount
end for
```

setup_generateHouseholds()

Assigns each household a type which gives the number of adults and children present. Each individual is then set up and assigned an appropriate age and workplace.

```
numberPeople←0
peopleArray.reserve(2.5 * numberHouseholds)
for hh=0 to numberHouseholds do
    numberAdults,numberChildren←assignHouseholdPopulation(hh)
    peopleArray.resize(numberAdults + numberChildren + numberPeople)
    hhIdx←numberWorkplaces + hh
    locationData[hhIdx].locationType← HOUSEHOLD
    for i←0 to numberAdults do
        peopleArray[numberPeople].Household←hhIdx
        peopleArray[numberPeople].Age←Adult
        peopleArray[numberPeople].Workplace←assignWorkplace()
        numberPeople++
    end for
    for i←0 to numberChildren do
        peopleArray[numberPeople].Household←hhIdx
        age ←assignChildAge()
        peopleArray[numberPeople].Age←age
        peopleArray[numberPeople].Workplace←assignSchool(age)
        numberPeople++
    end for
end for
```

setup_generateFixedLocations()

Generates workplace and household location arrays. Because these do not change throughout the simulation, they are generated once and re-used.

```
dailyLocationArray.workplaceArray.resize(numberWorkplaces)
dailyLocationArray.householdArray.resize(numberHouseholds + numberWorkplaces)
for personIdx←0 to numberPeople do
    personPtr←&peopleArray[personIdx]
    dailyLocationArray.workplaceArray.insert(personPtr.Workplace, personPtr)
    dailyLocationArray.householdArray.insert(personPtr.Household, personPtr)
end for
```

setup_initialInfection()

Selects individuals to initiate the outbreaks and transitions them to infected status.

```
victims[]←getUniquePeople(numberInitialInfections_pandemic)
for i← 0 to numberInitialInfections_pandemic do
    personPtr←victims[i]
    startSymptomaticInfection_pandemic(personPtr)
end for
victims[]←getUniquePeople(numberInitialInfections_seasonal)
for i←0 to numberInitialInfections_seasonal do
    personPtr←victims[i]
    startInfection_seasonal(personPtr)
end for
```

9.1.3 Daily Loop

updateStatus(personPtr**)**

Tests whether individuals have reached culmination, and if so updates their status.

```
if personInfected_Pandemic() and dayPandemicInfection + culminationPeriod == currentDay then
    recoverPandemic()
end if
if personInfected_Seasonal() and daySeasonalInfection + culminationPeriod == currentDay then
    recoverSeasonal()
```

```
end if
```

generateSchedule(personPtr)

Sets up agent's schedules for the current day, including errand assignment. Also marks how many contacts should be made during each hour.

```
for hour←0 to 23 do
  schedule[hour]← personPtr-¿Household
  contactsDesired[hour]← 0
  contactType[hour]← HOUSEHOLD
end for
if is_weekend() then
  errandHours[]←getThreeUniqueErrandHours()
  errandDestinations←getThreeErrandDestinations()
  contactsProfile← [2,0,0],[0,2,0],[0,0,2],[1,1,0],[1,0,1], or [0,0,1]
  for errand← 0 to 2 do
    hour← errandHours[errand]
    schedule[hour]←errandDestinations[errand]
    contactsDesired[hour]←contactsProfile[errand]
    contactType[hour]← ERRAND
  end for
else
  for hour←WORK_START to WORK_END do
    schedule[hour]← personPtr-¿Workplace
    contactType[hour]← WORKPLACE
  end for
  contactsDesired[WORK_START]← workplaceMaxContacts[personPtr-¿Workplace]
  if isChild() then
    afterschoolDestination← getAfterschoolDestination()
    schedule[AFTERSCHOOL_START]← afterschoolDestination
    schedule[AFTERSCHOOL_START + 1]← afterschoolDestination
    contactsDesired[AFTERSCHOOL_START]← 2
    contactType[AFTERSCHOOL_START]←AFTERSCHOOL
    contactType[AFTERSCHOOL_START+1]←AFTERSCHOOL
  else
    schedule[ERRAND_HOUR1]← getErrandDestination()
    schedule[ERRAND_HOUR2]← getErrandDestination()
```

```
            contactsProfile← [2,0],[0,2], or [1,1]
            contactsDesired[ERRAND_HOUR1]← contactsProfile[0]
            contactsDesired[ERRAND_HOUR2]← contactsProfile[1]
            contactType[ERRAND_HOUR1]←ERRAND
            contactType[ERRAND_HOUR2]←ERRAND
        end if
        contactsDesired[HOUR_HOME]← HOUSEHOLD_MAX_CONTACTS
    end if
```

insertToLocations(personPtr**)**

Inserts agents into location array to generate locations. As the household and workplace errands do not change throughout the simulation, they are generated once during setup and re-used. Therefore, this method only inserts the errand destinations.

```
    if is_weekend() then
        errand1← personPtr->schedule[ERRAND_HOUR1]
        dailyLocationArray.insert(ERRAND_HOUR1,errand1,personPtr)
        errand2← personPtr-¿schedule[ERRAND_HOUR2]
        dailyLocationArray.insert(ERRAND_HOUR2,errand2,personPtr)
    else
        for errand←0 to 2 do
            errandHour← personPtr-¿errandHours[errand]
            errandDestination←personPtr-¿schedule[errandHour]
            dailyLocationArray.insert(errandHour,errand,personPtr)
        end for
    end if
```

makeContacts(personPtr**)**

Iterates an individual's schedule, selecting contacts during hours marked by the schedule generation algorithm.

```
    contactIds[]←
    contactTypes[]←
    contactsMade←0
    for hour←0 to 23 do
        contactsdesired←personPtr-¿contactsDesired[hour]
        while contactsdesired>0 do
            location← personPtr-¿schedule[hour]
            contact← contactselectRandomPersonFromLocation(hour,location,personPtr)
            if contact == NULL then
                contactType←TYPE_NULL
            else
                contactType←personPtr-¿contactType[hour]
            end if
            contactIds[contactsMade]←contact
            contactTypes[contactsMade]←contactType
            contactsMade++
            contactsdesired–
        end while
    end for
```

processContacts(contactIds[ ],contactTypes[ ]**)**

Calibrates infection probabilities, and processes each contact to determine whether infection is successful. If so, the victim is transitioned to infected status and assigned a profile, etc.

```
weightSum←sumWeights(contactTypes)
infectionThreshold_pandemic←-1.0
infectionThreshold_seasonal←-1.0
if personInfected_Pandemic() then
    dayOfInfection←currentDay - personPtr-¿infectionDayPandemic
    profile←personPtr-¿profilePandemic
    infectionThreshold_pandemic←calculateInfectionThreshold(baseReproduction_pandemic, profile, dayOfInfection, weightSum)
end if
if personInfected_Seasonal() then
    dayOfInfection←currentDay - personPtr-¿infectionDaySeasonal
    profile←personPtr-¿profileSeasonal
    infectionThreshold_pandemic←calculateInfectionThreshold(baseReproduction_seasonal, profile, dayOfInfection, weightSum)
end if
for contact←0 to contactsMade do
    yval_p←randFloat(0.0,1.0)
    yval_s←randFloat(0.0,1.0)
    successPandemic← yval_p<infectionThreshold_pandemic * calcContactWeight()
    successSeasonal← yval_s<infectionThreshold_seasonal * calcContactWeight()
    if successPandemic then
      transmitInfection_pandemic(infector,victim)
    end if
    if successSeasonal then
      transmitInfection_seasonal(infector,victim)
    end if
  end for
```

calculateInfectionThreshold(baseRn,profile,dayOfInfection,weightSum**)**

Determines the probability of success for a contact with weight 1.0 based on the profile, day of infection, weight sum, and reproduction number. This probability can be further adjusted for individual contact weights.

```
  dailyFraction←sheddingProfiles[profile][dayOfInfection]
  asympAdjustment←((1.0 - asymp_reduction_factor)*percent_symptomatic) + asymp_reduction_factor
  prob← dailyFraction * baseRn / asympAdjustment
  if is_asymptomatic(profile) then
    prob *= asymp_reduction_factor
  end if
  return prob
```

9.1.4 Final Reproduction Calculations

calculateReproduction()

Calculates reproduction numbers and generation size for each generation, and then outputs to disk.

```
  generationSize_pandemic[MAX_DAYS]
  generationSize_seasonal[MAX_DAYS]
  for day←0 to MAX_DAYS do
    generationSize_pandemic[day]←0
    generationSize_seasonal[day]←0
  end for
  for i←0 to numberPeople do
    if peopleArray[i].StatusPandemic != STATUS_SUSCEPTIBLE then
      gen_p← peopleArray[i].GenerationPandemic
      generationSize_pandemic[gen_p]++
    end if
    if peopleArray[i].StatusSeasonal != STATUS_SUSCEPTIBLE then
      gen_s← peopleArray[i].GenerationSeasonal
      generationSize_seasonal[gen_s]++
    end if
  end for
  for gen←0 to MAX_DAYS do
    genSize_p← generationSize_pandemic[gen]
    nextGenSize_p← generationSize_pandemic[gen+1]
    genSize_s← generationSize_seasonal[gen]
    nextGenSize_s← generationSize_seasonal[gen+1]
    generationRn_p←genSize_p / nextGenSize_p
    generationRn_s←genSize_s / nextGenSize_s
    outputLine(gen,genSize_p,generationRn_p,genSize_s,generationRn_s)
  end for
```

9.2 GPU Algorithms

9.2.1 Main

main()

```
  setupSim()
  for day←0 to max do
    countAndRecover()
    assignSchedules()
    generateLocations()
    selectInfected()
    makeAndProcessContacts()
  end for
  calculateReproduction()
```

9.2.2 Setup

setupSim()

```
  setup_readInputs()
  setup_calcPopulationSize()
```

```
setup_sizeGlobalArrays()
setup_generateHouseholds()
setup_assignWorkplaces()
setup_initializeStatusArrays()
setup_initialInfected()
```

setup_calcPopulationSize()

Generates a typecode that gives number of adults and children for each household, and reduces these to produce the total population size. This allows arrays to be properly sized before households are generated.

```
for hh←0 to numberHouseholds do
    rngVal← getRngVal(hh)
    hhType← getHhType(rngVal)
    hhSize← hhAdultCount[hhType] + hhChildCount[hhType]
    REDUCE:+ hhSize
end for
```

setup_generateHouseholds()

Generates the same typecodes as before for each household, but stores them. The typecodes are then transformed to find the number of adults and children for each household and exclusive-scanned. For household i, this gives the total number of people in households 0 to $i-1$. This is used to assign household IDs to individuals, who are marked as either adults or children. A placeholder age code is used for children and will be replaced during workplace assignment. The household ID values of all individuals are stored in a lookup table. Since household locations remain the same throughout the simulation, they are generated here and re-used throughout. Since personIDs are assigned according to household, the array is already sorted, and the exclusive-scan values can be used directly rather than binary searching.

```
begin kernel:
        for hh←0 to numberHouseholds do
            rngVal← getRngVal(hh)
            hhType← getHhType(rngVal)
            yield hhTypeArray[hh]← hhType
        end for
end kernel
begin functor
```

```
    for hh←0 to numberHouseholds do
      hhType← hhTypeArray[hh]
      output hhAdultOffset[hh]← exclusive scan of hhAdultCount[hhType]
      output hhChildOffset[hh]← exclusive scan of hhChildCount[hhType]
    end for
  end functor
  begin kernel:
    for hh←0 to numberHouseholds do
      hhType← hhTypeArray[hh]
      adultCount← hhAdultCount[hhType]
      childCount← hhChildCount[hhType]
      hhOffset← hhAdultOffset[hh]+hhChildOffset[hh]
      output hhLocationOffset[hh]← hhOffset
      for i←0 to adultCount do
          personId← hhOffset + i
          output hhLookupArray[personId]←hh
          output peopleAgeArr[personId]← AGE_ADULT
      end for
      hhOffset← hhOffset + adultCount
      for i← 0 to childCount do
          personId← hhOffset + i
          output hhLookupArray[personId]←hh
          output peopleAgeArr[personId]← AGE_CHILD_PLACEHOLDER
      end for
    end for
  end kernel
  hhLocationOffset[numberHouseholds]← numberPeople
```

setup_generateWorkplaces()

Generates the workplace location array, which remains constant and is re-used throughout the simulation. For adults, a workplace is assigned according to the PDF. School assignment for children is according to their age code, so these steps are performed together. A location array is generated by sorting and binary searching for location offsets.

```
  begin functor:
    for personId←0 to numberPeople do
```

```
      myAge← peopleAgeArr[personId]
      if myAge == AGE_ADULT then
            output workplaceLocations[personId]←assignAdultWorkplace()
            output workplacePeople[personId]←personId
      else
            childAge,childSchool ← assignChildAgeAndSchool()
            output peopleAgeArr[personId]←childAge
            output workplaceLocations[personId]←childSchool
            output workplacePeople[personId]← personId
      end if
    end for
end functor
generateLocation(workplacePeople,workplaceLocations)
```

setup_initializeStatusArrays()

Sets all individuals to susceptible status, with null values for generation and day.

```
for personId←0 to numberPeople do
  output peopleStatusPandemic[personId]← STATUS_SUSCEPTIBLE
  output peopleStatusSeasonal[personId]← STATUS_SUSCEPTIBLE
  output peopleDayPandemic[personId]← DAY_NULL
  output peopleDaySeasonal[personId]← DAY_NULL
  output peopleGenPandemic[personId]← GEN_NULL
  output peopleGenSeasonal[personId]← GEN_NULL
end for
```

setup_initialInfected()

Selects and performs infection on initial population.

```
victims[ ]← nUniqueIndexes()
for i← 0 to initialInfectedPandemic do
  personId←victims[i]
  output peopleStatusPandemic[personId]← assignSymptomaticProfile()
  output peopleDayPandemic[personId]← INITIAL_DAY
  output peopleGenPandemic[personId]← 0
end for
for i←initialInfectedPandemic+1 to initialInfectedTotal do
  personId← victims[i]
```

```
    output peopleStatusSeasonal[personId]← assignProfile()
    output peopleDaySeasonal[personId]← INITIAL_DAY
    output peopleGenSeasonal[personId]← 0
end for
```

9.2.3 Daily Loop

daily_countAndRecover()

Transitions infected individuals who have reached culminated to recovered status, and counts the number of times each status code occurs in the population.

```
zero status counters
begin kernel:
    for personId← 0 to numberPeople do
        statusPandemic← peopleStatusPandemic[personId]
        dayPandemic← peopleDayPandemic[personId]
        if isInfected(statusPandemic) and currentDay - dayPandemic == CULMINATION_PERIOD
        then
            statusPandemic← STATUS_RECOVERED
            output peopleStatusPandemic[personId]← statusPandemic
        end if
        statusSeasonal← peopleStatusSeasonal[personId]
        daySeasonal← peopleDaySeasonal[personId]
        if isInfected(statusSeasonal) and currentDay - daySeasonal == CULMINATION_PERIOD then
            statusSeasonal← STATUS_RECOVERED
            output peopleStatusSeasonal[personId]← statusSeasonal
        end if
        pandemicStatusCounter[statusPandemic]++
        seasonalStatusCounter[statusSeasonal]++
    end for
    REDUCE:+ pandemicStatusCounter
    REDUCE:+ seasonalStatusCounter
end kernel
output S/I/R counts for pandemic and seasonal
```

daily_buildInfectedArray()

Selects individuals who are currently infected with one or more strains of virus. This is implemented using a select operator from the Thrust library using a custom predicate functor.

```
for personId←0 to numberPeople do
    statusPandemic←peopleStatusPandemic[personId]
    statusSeasonal←peopleStatusSeasonal[personId]
    if isInfected(statusPandemic) or isInfected(statusSeasonal) then
        yield personId
    end if
end for
result: array infectedIndexes, value infectedCount
```

doWeekday()

Sets up the weekday errand array and launches the weekday contact kernel.

```
begin kernel:
        for personId← 0 to numberPeople do
            outputOffset← 2 * personId
            output errandLocations[outputOffset, outputOffset+1]← assignErrandOrAfterschoolDestinations()
            output errandPeople[outputOffset,outputOffset+1]← personId
        end for
end kernel
sort value array errandPeople by key array errandLocations
binary search errandLocations for location ID numbers 0 to 2*numberLocations
launch weekday contact selection and processing kernel
```

doWeekend()

Sets up the weekend errand array and launches the weekend contact kernel.

```
begin kernel:
        for personId← 0 to numberPeople do
            outputOffset← 3 * personId
            output errandLocations[outputOffset,outputOffset+1,outputOffset+2]← assignWeekendErrandDestinations()
            output errandPeople[outputOffset,outputOffset+1,outputOffset+2]← personId
        end for
end kernel
sort value array errandPeople by key array errandLocations
binary search errandLocations for location ID numbers 0 to 10*numberLocations
launch weekend contact selection and processing kernel
```

kernel_doWeekdayContacts()

kernel_doWeekendContacts()

Launches the contact selection and processing methods. Each kernel invokes its respective contact selection kernel, but they are otherwise the same.

```
for arrayPos← 0 to infectedCount do
    personId← infectedIndexes[arrayPos]
    contactIds[ ]
    contactTypes[ ]
    makeContactsWeekday(personId,contactIds,contactTypes)
    processContacts(personId,contactIds,contactTypes)
end for
```

makeContactsWeekday(personId,contactIds,contactTypes**)**

Selects 3 contacts from household, 2 from errand or afterschool, and either 3 from workplace or 2 from school.

```
myHH← hhLookupArray[personId]
selectRandomPerson(myHH)
selectRandomPerson(myHH)
selectRandomPerson(myHH)
myWp← regenerateWP(personId)
numWpContacts← lookupWpMaxContacts(myWp)
while numWpContacts>0 do
    selectRandomPerson(myWp)
end while
errandDests[] ← regenerateErrandDestinations(personId)
profile←assignErrandContactsProfile()
for hour←0, 1 do
    errandContactsThisHour← errandContacts[profile][hour]
    while errandContactsThisHour>0 do
        selectRandomPerson(errandDests[hour])
    end while
end for
```

makeContactsWeekend(personId,contactIds,contactTypes**)**

Selects 3 contacts from household, and 2 from errands.

```
myHH← hhLookupArray[personId]
selectRandomPerson(myHH)
selectRandomPerson(myHH)
errandDests[] ← regenerateErrandDestinations(personId)
profile←assignErrandContactsProfile()
contact1_errandDest← errandContactHours[profile][0]
contact2_errandDest← errandContactHours[profile][1]
selectRandomPerson(contact1_errandDest)
selectRandomPerson(contact2_errandDest)
```

processContacts(personId,contactIds,contactTypes**)**

First, it calibrates infection probabilities from the expected number of daily contacts. The basic expected daily contact values for each day and each profile are initialized during setup, and are simply looked up. Each contact is then processed, and if successful the infection is transmitted to the selected victim.

```
weightSum← sumWeights(contactTypes)
infectionThreshold_pandemic←-1.0
infectionThreshold_seasonal←-1.0
statusPandemic← peopleStatusPandemic[personId]
genPandemic← peopleGenPandemic[personId]
if isInfected(statusPandemic) then
   profile← getProfileFromStatusCode(statusPandemic)
   dayOfInfection←currentDay - peopleDayPandemic[personId]
   infectionThreshold_pandemic←baseReproduction_pandemic * expectedDailyContacts[profile][dayOfInfection]
   / weightSum
end if
statusSeasonal← peopleStatusSeasonal[personId]
genSeasonal← peopleGenSeasonal[personId]
if isInfected(statusSeasonal) then
   profile← getProfileFromStatusCode(statusSeasonal)
   dayOfInfection←currentDay - peopleDaySeasonal[personId]
   infectionThreshold_seasonal←baseReproduction_seasonal * expectedDailyContacts[profile][dayOfInfection]
   / weightSum
end if
for contact←0 to contactsMade do
```

```
    yval_p←randFloat(0.0,1.0)
    yval_s←randFloat(0.0,1.0)
    successPandemic← yval_p<infectionThreshold_pandemic * calcContactWeight()
    successSeasonal← yval_s<infectionThreshold_seasonal * calcContactWeight()
    if successPandemic and victim is susceptible then
        peopleStatusPandemic[victim]←assignInfectedProfile()
        peopleGenPandemic[victim]← genPandemic+1
        peopleDayPandemic[victim]← currentDay+1
    end if
    if successSeasonal and victim is susceptible then
        peopleStatusSeasonal[victim]←assignInfectedProfile()
        peopleGenSeasonal[victim]← genSeasonal+1
        peopleDaySeasonal[victim]← currentDay+1
    end if
end for
```

selectRandomPerson(myLoc,locOffsets,locationPeople**)**

Selects a random person from a location who is not the infector. If only one person is at this location, no contact is possible and a null contact is stored.

```
lowerBound← locOffsets[myLoc]
peopleAtLoc← locOffsets[myLoc+1] - lowerBound
if peopleAtLoc==1 then
    output contactType← NULL_CONTACT
else
    personSelected← UnsignedRand() mod peopleAtLoc
    output contactId← locationPeople[lowerBound+personSelected]
    if contactTarget[i]==myIdx then
        personSelected← (personSelected+1) mod peopleAtLoc
        output contactId ← locationPeople[lowerBound+personSelected]
    end if
end if
```

9.2.4 Final Reproduction Calculations

calculateReproduction()

Finds the number of people in each generation by using a histogram algorithm. Then, calculate the reproduction for each generation and output to disk.

```
sort(peopleGenPandemic)
binary search for offset of generation 0 to MAX
sort(peopleGenSeasonal)
binary search for offset of generation 0 to MAX
for gen← 0 to MAX do
    genSizeP← genOffsetP[gen+1] - genOffsetP[gen]
    genSizeS← genOffsetS[gen+1] - genOffsetS[gen]
    nextGenSizeP← genOffsetP[gen+2] - genOffsetP[gen+1]
    nextGenSizeS← genOffsetS[gen+2] - genOffsetS[gen+1]
    genRn_P← nextGenSizeP / genSizeP
    genRn_S← nextGenSizeS / genSizeS
    output(gen,genSizeP,genRn_P,genSizeS,genRn_S)
end for
```