Acceleration of Agent-Based Pandemic Modeling on Multiple GPUs

Shekh

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
Acceleration of Agent-Based Pandemic Modeling on Multiple GPUs

by

Barzan Shekh

A thesis submitted to the Graduate College in partial fulfillment of the requirements for the degree of Master of Science
Department of Computer Science
Western Michigan University
August 2015

Thesis Committee:
Elise de Doncker, Ph.D., Chair
John Kapenga, Ph.D
Diana Prieto, Ph.D
Epidemiology computation models are crucial for the assessment and control of public health crises. Agent-based simulations of pandemic influenza are useful for forecasting the infectious disease spreading in order to help public health policy makers during emergencies. In such emergencies decisions are required for public health preparedness in cycles of less than a day, and the agent-based model should be adaptable and tractable for quick and simple calibration with low computational overhead.

GPU accelerated computing involves the use of a graphics processing unit (GPU) in combination with the CPU to perform heterogeneous computing by offloading a compute-expensive portion of the program to the GPU while the remaining program is running on the CPU. This thesis modifies former models considerably, explores the performance of a low-complexity agent-based model for pandemic simulations when accelerated by multiple GPUs on a single node computer.

In this thesis, we demonstrate the utilization of the hardware environment and software tools and discuss strategies for adapting agent-based simulation to multiple GPUs. We further compare the performance of simulations using two GPUs or four GPUs with the sequential execution on the CPU, in terms of time and speedup. The multi-GPU implementations exhibit great performance and support populations with up to 100 million individuals.
Acknowledgements

I would like to express my sincere gratitude and appreciation to my advisor, Prof. Elise de Doncker, for her continuous support, patience, and motivation in my academic study and this thesis research. Additionally, I would like to extend my gratitude to Prof. Diana Prieto for her help, guidance, enthusiasm, and background knowledge about the field of study.

Further, I would like to give special thanks to Prof. John Kapenga for his encouragement, support, guidance and insightful comments on the thesis.

Lastly, I would like to thank my family for their continuous support and for encouraging me in all of my pursuits and inspiring me to follow my dreams. I would like to thank my friends who helped me during my study at Western Michigan University.

Barzan Shekh
# Contents

Acknowledgements ii  
List of Figures v  
List of Tables vi  

1 Introduction and Objectives 1  
  1.1 Introduction ............................................. 1  
  1.2 Objectives ............................................. 3  

2 Literature Review 4  

3 Computational Model 7  
  3.1 Model .................................................. 7  
  3.2 Agents ................................................. 8  
  3.3 Epidemiological Model ............................... 10  
  3.4 Contact Network ....................................... 10  
  3.5 Parameter Calibration ................................. 11  

4 Hardware and Software Environment 13  
  4.1 Graphics processing unit computing (GPU) Computing .... 13  
  4.2 CPU vs GPU ........................................... 14  
  4.3 Hardware Architecture .................................. 16  
  4.4 OpenMP ............................................... 18  
  4.5 CUDA Programming Model ............................ 19  
  4.6 Memory ................................................ 20  
    4.6.1 GPU Memory .................................... 20  
      4.6.1.1 Global Memory ............................ 20  
      4.6.1.2 Register Memory ......................... 21  
      4.6.1.3 Local Memory ............................ 21  
      4.6.1.4 Constant Memory ......................... 21  
      4.6.1.5 Shared Memory ............................ 21
List of Figures

3.1 Agent Status During Simulation [1] .......................... 8
3.2 Generational Reproduction Number [1] ....................... 9

4.1 CPU and GUP Cores ........................................ 15
4.2 GK110 chip SMX Architecture [2] .......................... 16
4.3 Simplified Sandy-Bridge architecture ........................ 17
4.4 Shared Memory System ........................................ 18
4.5 General Syntax of kernel launch ............................. 19
4.6 Mapped Pinned Memory ....................................... 23
4.7 Portable Pinned Memory ....................................... 24
4.8 Write-Combined Pinned Memory ............................... 25

5.1 Generate Agent Sequential Pseudocode ...................... 27
5.2 Sequential Diagram .......................................... 28
5.3 Compute population size and generate intermediate arrays ... 31
5.4 Extracting array .............................................. 32
5.5 Assign_Agent kernel ........................................... 33
5.6 Send_AgentToLocations ....................................... 34
5.7 Splitting Agent into blocks and launching kernel on multiple GPUs 35

6.1 Runtime Comparison Figure ................................. 37
6.2 Speedup ...................................................... 39
List of Tables

6.1 Simulation Runtime Comparison in (sec) . . . . . . . . . . . . . . . 37
6.2 Speedup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
Chapter 1

Introduction and Objectives

1.1 Introduction

Over the past years researchers have shown that there is a need for adequate computer simulation models for public health preparedness (PHP) policies during pandemic influenza. Dreadful outbreaks such as the 1918 flu pandemic (Spanish flu), and H1N1 (swine flu) require a policy response from public health officials to help recommend the correct course of action. The capability of such computer simulation models to be able to adapt to current data, correctly simulate the problem and provide the right response has caused the Centre for Diseases Control and Prevention (CDC) in the U.S. to act upon this by creating a Preparedness modeling unit in 2006 [3]. These computer simulation models, over the past years, have been used by many federal agencies such as (CDC) to create the U.S. Community Containment Guidance for Pandemic Influenza. These existing pandemic influenza (PI) models have been well recognized. However, it has become questionable regarding whether or not these models can be used in operational decision making, and this has not yet been put into practical use by the state or by other public health practitioners. In emergency situations where the spread of a disease
increases, models must be able to provide statistically accurate results as quickly as possible since effective response in such situations becomes more difficult. These decisions must be made in less than a day [4].

Prieto et al. demonstrated several criteria that lead to influenza models applicable for operational decision-making [3]. First, models must include accurate field observation data of an epidemiological and demographical nature during the outbreak. The assumptions of the model ought to support precise decision making in the desired region, and moreover, must also be computationally tractable in order to support fast processing. Simulations must run sufficiently fast and be scalable to support large populations.

Generally, there are two types of simulation models: Diffusion-based models which divide the population into compartments (groups of people) based on the geographical area. These type of models offer quick results. However, they cannot implement fine-grained intervention strategies because the behavior of individuals is ignored. Agent-based models are designed to forecast the spread of the disease through interaction between specific individuals in the contact network. Parallel systems are used for these models because they are computationally expensive and require large amounts of memory.

GPU accelerated computing involves the use of a graphics processing unit (GPU) in combination with the CPU to perform heterogeneous computing by offloading a compute-expensive portion of the program to the GPU while the remaining program is running on the CPU. However, good performance depends on successful use of the GPU architecture, and not all algorithms are suitable for GPU acceleration.
1.2 Objectives

In this thesis, we introduce a high-performance framework of a low-complexity agent-based model for pandemic simulation and support populations with up to 100 million individuals when accelerated by multiple GPUs on a single node computer. In chapter 2, we review the literature on existing frameworks for influenza simulations. Chapter 3 describes the baseline of computational and epidemiological models that are used in this research. In chapter 4 we demonstrate the utilization of the hardware environment and software tools. We further present the baseline of our novel sequential implementation and discuss strategies for adapting agent-based simulation to multiple GPUs in chapter 5. Finally, chapter 6 gives performance results and compares the performance of simulations using two GPUs and four GPUs with the sequential execution on the CPU, in terms of time and speedup.
When modeling epidemics there are generally two types of models: population-based and agent based models. In population-based modeling, the population is divided with respect to cities or large sub areas that are used as compartments. Individuals in the same compartment have the same infectious probability. The disease spreading is then modeled by equations that describe the population transitions between these compartments. The population-based approach usually produces rapid results. The disadvantage of using such models is the low utility of operational modeling due to the fact that they are not based on individuals but rather on compartments and as such cannot implement fine-grained intervention strategies.

There are many population-based models available such as the Global Epidemic Model described in [5], and the SpatioTemporal Epidemiological Modeler (STEM), which is an open source Eclipse software. This software uses differential equation such as finite difference Runge-Kutta 4th order to simulate disease spread among compartments based on transportation and the boundaries of neighboring compartments [6]. The Global Epidemic and Mobility Model (GLEAM) is a computational tool software that executes on the GLEAM project’s high performance computing cluster and explores realistic epidemic spreading scenarios at the global
scale. GLEAMviz which is the client side that can be installed on personal computer and connects to the GLEAM server. GLEAMviz allows users to build and visualize epidemic models that are running on the GLEAM cluster [7].

The agent-based approach it models the behavior of individuals known as agents. The disease spreading in such a model is controlled by the model networks. Agents follow a specified schedule; this schedule specifies when/where they may or may not make contacts with other individuals. The contact network is generated based on the agent’s schedule and the contacts made by the agents at certain locations. This network is then used to further spread the disease between agents. The advantage of this type of model is that it is ideal for operational modeling due to the fact that interventions can be used to adjust the contact network as well as the disease transmission probabilities for individuals.

The Self-Calibrating agent-based method is an individual-based model that simulates pandemic and seasonal influenza for up to fifty million individuals and produces quick results by harnessing the power of general-purpose computing on graphics processing units. This simulation is developed to work on a personal computer or workstation computers that have an NVIDIA graphics processing unit [4]. EpiSimdemics an agent-based model that uses a rather complex contact model. In this model susceptible individuals have the chance of becoming infected every time they visit a certain location. Whether or not they get infected depends on many factors, some of which are the viral shedding of present individuals, the susceptibility of the individual and the total amount of contact with the present infectious individuals. Many factors have to be taken into account in order to compute the contact network accurately, which increases computational complexity. The contact network and the epidemiological model play an important role in the runtime of the simulation as well as the epidemiological accuracy of the simulation. There is another agent-based model software for epidemic influenza called FluTE, which is developed in C/C++ language. This software takes about
two hours for simulating an unmitigated epidemic in a population of ten million [8].

Since such models are of high complexity it is required that they are processed on a (distributed memory) cluster due to memory and/or time requirements. In order to do this, the program is run as a set of sub programs mapped to processes, which are then distributed to multiple nodes within the cluster environment. Each node has access only to its own memory, and hence collective communications must be used across all nodes. MPI (Message Passing Interface) tools are generally used to implement this. The advantage of using this strategy is that many computer resources can be pulled together to take on the problem. However, the disadvantage is the greater burden put on the user as now the user must determine a suitable partitioning and proper communication schemes for the problem. Another disadvantage follows from the expense of accessing this type of cluster environment or even the unavailability of such a system. This strategy is used in most of the published work including EpiSimdemics as well as FluTE and many other agent-based frameworks such as ABM++ [9].
Chapter 3

Computational Model

3.1 Model

The simulation we have implemented is an agent based model of co-circulating influenza strains, pandemic (PI), and seasonal (SI), within the human population. The simulation emulates the spreading of the disease by daily interactions of individuals. This model has been adapted from Prieto et al. [10] and is explained in this section.

Agents in this model transition between three different states: Susceptible, Infected and Recovered states. All agents are initially in the Susceptible state. A random selected population of agents is given the status of infected so as to initiate the outbreak. The agents with an infected status make contact with other agents and start spreading the disease. Each agent has a schedule which identifies its location every hour. The spreading of the disease occurs at certain hours where the infected agents make contact with random susceptible agents at a certain location. By the end of the day new infected agents appear from contact with other previously infected agents. This process is repeated every day. The simulation runs every day hour by hour where each agent is placed in a certain location.
where they make contact randomly with other agents. Infected agents transition to the Recovered state after a culmination period of 10 days. The status of each agent is maintained separately for both pandemic and seasonal strains of the virus.

![Figure 3.1: Agent Status During Simulation [1]](image)

### 3.2 Agents

At the beginning of the simulation each individual is assigned to household, age group, a school or workplace, and a set of errand places. These individuals will visit these places at certain times in the day. The data used in this simulation is based on data from Hillsborough County in Tampa, Florida, with information from the 2002 U.S. Economic Census, the 2001 American Community Survey and the 2001 National household Travel Survey [10].

In order to be able to accurately simulate the transmission of the disease, a realistic representation of human contact must be available. These contact networks are generated from the individuals present at the infected agent’s location. A
scheduling algorithm is used to model human contacts. This scheduling algorithm determines the locations where the agents will be throughout the day.

This model mimics a typical day of an individual. From 8am to 5pm an adult is scheduled to go to work whereas a child is scheduled to go to school. In the hours between 5pm and 7pm an adult is scheduled to run two randomly chosen errands at two locations whereas children will attend a randomly selected after-school activity. During the weekends random individuals are selected to run errands at three different locations between 9am and 7pm, and other individuals will remain at home.

![Figure 3.2: Generational Reproduction Number](image)

The simulation could run for 24 hours to represent one day, however, to increase performance only the hours at which contact is made need to be simulated. Hence, during the weekdays only the time periods at work, running errands and home are simulated. During the weekends 10 hours are simulated, which represent the 10 possible errand hours between 9am and 7pm and home contact.
3.3 Epidemiological Model

A basic reproduction number is set as an input parameter. This number represents the average number of possible infections and how many other an infected individual can infect throughout their culmination period, assuming the entire population is susceptible. Another parameter used is the viral shedding, which is the amount of spreading of the virus that occurs between infection and recovery of each infectious individual everyday, and is represented as a fraction of the total viral shedding throughout this period. This value along with the basic reproduction number is used to distribute the expected number of infections per day. The probability of successful transmission to each contact is then found by further distributing the expected daily reproduction number through the daily contacts of an infected case [1].

A reproduction number is calculated for each generation of each strain after running the simulation for a certain number of days. The model is assumed to be calibrated when the basic reproduction number $R_0$ is statistically similar to the maximum value of the reproduction number obtained across all the generations. The reproduction number $R_0$ for generation $k$ is found as the ratio between the number of infected cases in generation $k+1$ and the number of infected cases in generation $k$ [10].

3.4 Contact Network

Infected contacts visit random locations and make contact with other individuals. At each contact they make, random individuals are selected as possible opportunities to spread the disease. The activity of the agent determines the possibility of transmitting and spreading the disease. The number of contacts made is based on a survey from European countries by [11]. The survey states that 23%
of contacts are made at home, 21% at work, 14% at school and 16% during leisure activities. It also gives the average number of contacts, which ranges between 7.95 and 19.77 depending on the country. During the weekdays more contacts are made than over the weekend. No similar data is found for America so the assumption is made that European countries are similar. Prieto et al. [10] report calculations based on the survey of Mossong et al. [11], which finds the number of contacts made. During weekdays each adult will select 3 contacts at the workplace, 2 contacts while running errands and 3 contacts in their household. Each child will select 2 contacts during school time, 2 contacts at their after-school activities and 3 contacts in their household. Both children and adults will select 2 contacts while running their errands and 3 contacts in their household. The selection of who the contacts are is performed randomly from the individuals found at the same location as the infectious individual, disregarding the status of the selected individual. If the same individual is chosen as the contact it is disregarded and the next contact at the same location is used instead. If at that location there was only one individual present, the infectious individual cannot select any other contact. In this case a null contact is stored [1].

3.5 Parameter Calibration

The epidemiological model has some features that distinguish it from other past models. For instance, the rate of infection in past approaches cannot be directly controlled; instead it is a second-order result of epidemiological parameters. When a susceptible agent visits certain locations they will have a certain probability of being infected. This probability is calculated based on different factors such as the duration of exposure to viral shedding and agent susceptibility. Disease reproduction must match the observed field data and, in order to do so, the parameters must be calibrated. This calibration is done using an optimization process [1].
The optimization process usually increases the model’s complexity as well as the model’s computational cost. Our model attempts to reduce the complexity by modeling the transmission from the perspective of infected individuals rather than modeling the exposure of susceptible individuals to the virus. Transmission from infectious individuals to other individuals within their contact network is modeled. The basic reproduction number, which is the average of infected individuals per case, is given from the observational field data. The probability of successful transmission for each individual within the contact network is assigned using the reproduction number [1].

Viral shedding is used to model the infectiousness of an individual as this can vary throughout the period of infection. Measurements of the viral infection are provided by Carrat et al [12] for each day of the period of infection. By summing all these measurements we can get the total amount of viral shedding for the whole period and we can calculate the fraction of the total viral shedding that occurs between infection and recovery each day. The expected reproduction value for each day is then calculated by using the fraction of viral shedding and distributing the calibration reproduction number across the period of infectiousness.

Each contact made within the contact network is then assigned a probability of transmission using the expected daily reproduction number, assuming that the each susceptible contact is equally probable to get infected and the number of infected cases follows a binomial distribution. A Kappa value is then assigned to each contact to adjust the probability of infection based on the type of activity, this is done so that contacts can be represented with different probabilities of success [1].
Chapter 4

Hardware and Software Environment

4.1 Graphics processing unit computing (GPU) Computing

The current extraordinary revolution in scientific and engineering computation is happening as a result of the massive multi-threading computation in computer graphics which is pioneered by NVIDIA. This has evolved into general-purpose computing on graphics processing units (GPGPU computing). NVIDIA introduced CUDA (Compute Unified Device Architecture), which is an extension of the C language [13]. CUDA allows GPU code to be based on regular C, to offload computationally expensive work without having to resort to more complex shader languages. NVIDIA supports three different CUDA GPU architectures, namely Tesla, Fermi, and Kepler each of which has various generations. Different generations have different numbers of streaming multiprocessors (SM), CUDA Cores, memory size, memory bandwidth, memory clock rate as well as different graphics clock frequencies and processor clock frequencies.
4.2 CPU vs GPU

In order to maintain the execution speed of sequential programs in old central processing units (CPU) with a single-core processor architecture, processor manufacturers were increasing the clock rate of the processor. In modern processors, the clock rate increase has hit a limit for single-core processors at around 4GHz. This limit is due to the fact that CPUs at very high frequencies generate too much heat, which requires advanced and expensive cooling systems. As we increase the clock rate, the power consumption rises. The limitation of the clock rate has forced processor manufacturers to adopt a different approach. They settled on a multicore trajectory, which is designed to maximize the execution speed of sequential programs by adapting to multiple cores instead of continuously increasing the clock rate of a single core. The first generation of multi-cores began with just two-core processors. With each semiconductor generation the number of cores increased to four, six, eight and sixteen. For example Intel, the Xeon E5 2670 (Sandy-Bridge processor), which is widely used for servers and high performance (HPC) computing cluster nodes, has eight cores and also supports hyperthreading. In our local cluster "thor", most of the compute nodes have dual 8-core Xeon E5-2670 processors (for a total of 16 cores per node).

In contrast, GPUs deploy many-core technologies that focus more on the execution throughput of the parallel applications with a low processor clock rate. GPUs have a single, or multiple streaming multiprocessors (SMs); each SM contains a number of streaming processors (SPs) or CUDA cores. The numbers of SMs and cores vary from generation to generation. Most of the compute nodes in our local "thor" cluster have NVIDIA Tesla K20 GPUs. The Tesla K20 GPUs contain one NVIDIA GK110 chip, which is one of the most efficient HPC architectures. The GK110 contains 15 new streaming (SMX) multiprocessors. Each SMX has 192 cores, 64KB shared memory L1 cache and 48KB read-only data cache. Each SMX can execute up to 2048 threads concurrently. Figure 4.1 shows the
GK110 layout compared to that of a CPU. Figure 4.2 shows the SMX architecture in detail. An architecture with such capabilities is perfectly suited for algorithms that are embarrassingly parallel and computationally intensive. It also allows programmers to apply heterogeneous computing in order to gain performance by taking advantage of both the GPU and the CPU. This works by transferring and executing the computationally intensive part of the application in parallel on the GPU while the CPU executes the rest of the application sequentially. Figure 4.1 highlights the differences between CPU and GPU.

Memory bandwidth is another difference between CPU and GPU in terms of performance. The maximum memory bandwidth of the Xeon E5 2670 is 51.2 GB per second. On the other hand, the maximum memory bandwidth of the Tesla K20 HPC graphics card with one GPU GK110 is 208 GB per second for data transfer between the GPU internal global memory and the GK110 chip.

![CPU and GPU Cores](image)

**Figure 4.1:** CPU and GPU Cores
4.3 Hardware Architecture

It is very important for a CUDA developer to understand the hardware configuration of the system before programming the application, because there are different CUDA programming approaches for different CPU/GPU configurations. For instance, the design of the simulation in this paper is based on the shared
memory structure of a node of the HPC cluster in Western Michigan University’s Parallel Computing and Data Science (PCDS) Laboratory. This node has dual Sandy-Bridge Xeon E5 processors with 128 GB of RAM and four Tesla K20 GPU accelerators as shown in Figure 4.3. In Intel Sandy-Bridge class processor the I/O hub is integrated into the CPU. A single Sandy-Bridge CPU has up to two QuickPath Interconnect (QPI) channels, used for accessing remote memory that is connected to another socket. It also has up to 40 lanes Gen. three PCI Express bandwidth. Note that the Tesla K20 has up to 16 lanes so 40 lanes of PCI Express are sufficient for two full size GPUs. Integrated PCI-E has advantages and drawbacks for CUDA programming. The drawback is that the PCI-E traffic is always affinitized, thus in Multi-CPU systems, GPUs associated with different CPUs cannot perform peer-to-peer operations. On the other hand, the CPU cache can participate in PCI-E bus traffic; the CPU can service Direct Memory Access (DMA) read request out of cache, and writes by the GPU are posted to the CPU cache. For Sandy-Bridge Multi-CPU configurations, pinned memory is preferred over the peer-to-peer operations [14].

Figure 4.3: Simplified Sandy-Bridge architecture
4.4 OpenMP

OpenMP is an application programming interface used for shared-memory parallel programming. The MP signifies ”multi-processing” in OpenMP, where it corresponds with multi-threading. OpenMP is designed for systems in which each thread has access to all available memory. In OpenMp programming, the system may be viewed as a collection of CPU cores, where all have access to main memory.

![Shared Memory System](image)

**Figure 4.4: Shared Memory System**

OpenMP sometimes permits the programmer to assign a block of code to be executed in parallel, and the actual parallelization of the given block is handled by the compiler and run-time system. OpenMP contains a suite of compiler directives as an extension to C/ C++ and Fortran for shared-memory multi-threaded programming.

The compiler directives called "pragmas" in C are special preprocessor instructions, to enable behaviors that are not part of the basic C specification. Pragmas may be ignored by the compiler if they are not supported. Thus, a program can be executed and still produces the right results even if some pragmas are not supported by the platform [15].
4.5 CUDA Programming Model

The CUDA programming model is based on a set of application programming interface (API) tools as an extension to the standard C language and Fortran, that allow programmers to perform parallel computations on NVIDIA CUDA enabled GPUs. NVIDIA provides CUDA with a handy CUDA software development kit (SDK), which includes the compiler, debugger, profiler, and runtime API, etc. There are some alternatives to CUDA for GPUs such as OpenCL and DirectCompute; however, we chose the CUDA framework.

In the CUDA programming model, the computing system consists of a host and one or more devices. The host refers to the CPU and its memory, while the device refers to a GPU and its memory. The code that run on the host side can manage memory on the host and device. In an ordinary CUDA program, the host code will allocate a certain size of memory on the host and the device memory, copy the data from the host to the allocated device memory, launch one or more kernels to perform parallel computations on the device, and finally copy the results back form the device to the host.

A kernel is similar to a C function; however, the start of a kernel in the program is indicated by a function qualifier `__global__` keyword. It is launched as grids of blocks of threads form the host and invoked using triple-angle-bracket in the call as follows:

\[
\text{Kernel}<<<\text{Grid\_Size, Block\_Size, Shared\_Memory, Stream}\gg\gg(\text{Parameters, }...);
\]

*Figure 4.5: General Syntax of kernel launch*

kernels are executed on the device; furthermore, for devices of compute capability 3.0 or higher kernels can be launched from another kernel for the purpose of Dynamic Parallelism. A kernel executes asynchronously meaning that it returns before the device has finished its execution. The grid size specifies the size of
an array of blocks; it can be one or two dimensions for compute capability 1.X hardware, or up to three dimensions for compute capability 2.X and higher. The block size specifies the dimension of the thread blocks which can be one, two, or three dimensions. All blocks in the grid have the same size; the block size may be up to 512 or 1024 threads. Thread blocks are scheduled separately onto SMs, and threads within the same block are executed by the same SM. Each SM splits its thread block into chunks of 32 threads called warps and the threads in a warp (with thread ID called “lane” within the warp) are executed simultaneously in a single instruction, multiple data (SIMD) fashion. [14].

4.6 Memory

Different types of memory are used by CUDA to maximize performance. The type of memory used depends on the expected utilization. Device memory attached to the GPU is by far the biggest GPU memory. Measured in gigabytes, it can be allocated and accessed in different ways. Since device memory is located on the GPU board, it can be accessed by an integrated GPU memory controller. Host memory refers to the random access memory (RAM) that is attached to the CPU(s) in the system. There are CUDA APIs are available to enable faster access to host memory by page-locking and mapping for the GPU.

4.6.1 GPU Memory

4.6.1.1 Global Memory

Global Memory can be allocated statically by using _device_ keyword in front of the memory deceleration, or dynamically using cudaMalloc(), and destroyed using the cudaFree() function. Data can be copied from the host to the
device memory or from the device to the host via cudaMemcpy(). Kernels can perform read/write operations on allocated global memory via pointers.

4.6.1.2 Register Memory

Register Memory is located within the SM; it is visible only to the thread that wrote it during the lifetime of that thread.

4.6.1.3 Local Memory

Local Memory is actually a memory concept utilized exclusively by the compiler. It contains the stack of local variables that cannot reside in registers (register spilling), parameters, and return addresses for subroutines or functions. Local memory is not an actual hardware component; instead it resides on global memory.

4.6.1.4 Constant Memory

Constant Memory is read-only memory that resides on device memory. It is optimized for read-only broadcast to multiple threads when all threads are read from the same location. Constant memory can be declared with the _constant_ keyword. Data can be copied to and from constant memory using cudaMemcpyToSymbol() and cudaMemcpyFromSymbol(), and even the pointer of _constant_ can be queried using cudaGetSymbolAddress().

4.6.1.5 Shared Memory

Shared Memory is an on-chip memory that is implemented inside SMs. Shared memory is 10x faster than global memory and around 10x slower than registers, it can be accessed very quickly. It is used to exchange data between threads within the same block. Shared memory can be declared using the _shared_ keyword.
4.6.1.6 Texture Memory

Texture memory in CUDA is realized in two parts: a CUDA array containing the physical memory allocation, and texture reference or surface reference containing the view that may be used to read/write a CUDA array. This type of memory is optimized for 2D spatial locality, and performance can be increased when all the lanes in a warp access nearby locations in texture memory according to expectation of locality [14].

4.6.2 Host Memory

Host memory by default is pageable, meaning the memory may be expelled out to disk by the operating system. Peripherals like GPUs cannot access host pageable memory because the physical addresses of this type of memory may change without notice. For enabling direct memory access (DMA) by hardware, operating systems allow host memory to be page-locked. CUDA, for performance reasons, includes APIs that make page-locked operating system resources available to application developers. This is commonly known as pinned memory that has been page-locked and mapped for direct access by CUDA enabled GPUs [14].

Pinned Memory is allocated through a CUDA runtime special function called cudaHostAlloc(), and freed by the cudaFreeHost() function. These functions work with the operating system to allocate page-locked memory and map it for direct memory access by the GPUs. CUDA monitors memory that has been allocated and boosts memory copies that involve host pointers allocated by cudaHostAlloc(). Default pinned memory is used as a buffer to increase data transfer performance, and yet it does not enable device kernels to access host memory. There are some features that allow device kernel access of pinned memory, three of these features are used in this thesis, are explained in subsequent subsections.
4.6.2.1 Mapped Pinned Memory

Mapped Pinned Memory is page-locked host memory that has been mapped into the CUDA address space, which permit CUDA kernels to directly access and perform read/write operations on it as illustrated in Figure 4.6. This type of host memory is allocated using cudaHostAlloc() with the cudaHostAllocMapped flag, and it is only accessible to the kernels that are being executed on the current GPU when cudaHostAlloc() is called. The GPU and CPU have updated address ranges that point to the same host memory buffer in their page table. Since the GPU has its own address space, the device pointers to the buffer have to be queried using cudaHostGetDevicePointer().

![Figure 4.6: Mapped Pinned Memory](image)

4.6.2.2 Portable Pinned Memory

As mentioned before, pinned memory is available to the GPU whose context is current, although by invoking cudaHostAlloc() with the cudaHostAllocPortable flag, the allocated page-locked host memory will be mapped into all the CUDA address spaces. A separate set of page table entries is created, and the page-locked host memory address range is mapped for all GPUs plus the CPU in the system.
Similar to mapped pinned memory, GPU pointers to the portable host memory buffer must be queried using cudaHostGetDevicePointer(). It is good practice to assign all pinned allocations as portable for applications that intend to use multi-GPUs, especially when Peer-to-Peer operations (reading, writing, copying, and streaming data that are located on the GPU’s global memory by another GPU in the system) is not allowed among all the GPUs [14].

Figure 4.7: Portable Pinned Memory

### 4.6.2.3 Write-Combined Pinned Memory

Write-Combined memory also known as Uncacheable Write Combining (USWC) memory was first created by Intel to enable the CPU to write to the GPU’s host memory buffer. In CUDA, write combined pinned memory can be declared by invoking cudaHostAlloc() with the cudaHostWriteCombined flag [14]. Application developers can perform Zero-Copy, by using the mapped or portable flag with the write-combined flag for pinned memory.
Figure 4.8: Write-Combined Pinned Memory
Chapter 5

Implementation

5.1 Traditional Sequential Approach

In this approach the simulation, implemented in the C-language, which is executed sequentially on the CPU as a standard reference. The sequential implementation is used for the purpose of computational model accuracy as well as for performance comparisons with the parallel implementations. There are three primary objects, agent, household, and business, which are implemented as structures. Using malloc(), objects of these structures are allocated as an array of structures (AOS). The simulation program reads the given data from files, and generates an array of ten thousand random numbers. Later on these random numbers are picked in a round-robin fashion and used throughout the simulation. There are several types of businesses with different numbers of each particular type as well as different contact rates. In function Generate_businesses(), nested loops are used to run through all the business types and initialize each object of the business array.

Agents are generated within the households, Figure 5.1 lists pseudo-code of the Generate_Agents function. There are different types of households based on
the household members. Household is generated randomly by using a uniform random number in the interval (0,1) and comparing it with the probability mass function of the household types. The nominated household will initialize \( n \) Adults and \( m \) children, then assign the workplace for each adult and school for each child. Adults and children are stored as Agents in the Agent array. This process is repeated in a loop over the given number of households. A counter that is kept total number generated of agents.

\[
\text{Generate\_Agents}() \\
\{ \\
\quad \text{for } i = 1 : \text{to number\_population\_center} \\
\quad \quad \text{for } k = 1 : \text{to number\_household} \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{R1} = \text{uniform}(0,1); \\
\quad \quad \quad \quad \text{for } s = 1 : \text{to number\_household\_type} \\
\quad \quad \quad \quad \quad \text{if } \text{R1} \geq \text{households}[s-1][3] \&\& \text{R1} < \text{households}[s][3] \\
\quad \quad \quad \quad \quad \text{Adult} = \text{households}[s][1]; \\
\quad \quad \quad \quad \quad \text{Child} = \text{households}[s][2]; \\
\quad \quad \quad \quad \} \\
\quad \quad \quad \} \\
\}
\]

\( \text{Figure 5.1: Generate Agent Sequential Pseudocode} \)

For the initial infection, \( l \) agents are picked randomly to contract pandemic flu. Furthermore, \( l \) agents are chosen randomly form the agent array to contract seasonal flu, and the remaining agents are susceptible. The daily loop is executed for all simulated agents. At the beginning of every day, a schedule of 24 locations
is assigned to each agent, where the agent will be present at each hour. The disease status is monitored hourly for every agent. Agents whose infection has reached culmination (after a duration of 10 days) are transitioned to recovered status. Agents are sent to their scheduled location in each hour. Infected agents will choose contacts randomly from the location’s agent network, leading to potential disease transmission (as explained in Section 3.4). Figure 5.2 depicts the program flow of the sequential implementation.

![Sequential Diagram](image-url)

**Figure 5.2: Sequential Diagram**
5.2 GPU Approach

Our approach leads to a parallelized version of the model that can be executed on multiple GPUs using CUDA. Implementing GPU acceleration has vital challenges, arising from the dynamic nature of the simulation and possible race conditions on the GPUs. In order to improve the performance of the agent-based simulation, the simulation functions are implemented as CUDA kernels for execution on one or multiple GPUs.

Portable pinned memory (see Section 4.6.2.2) is used to allocate an array of structures on the host memory for the primary objects (agent, business, household) in the simulations for the following reasons:

1. Peer-to-Peer operation is not allowed between GPUs that are connected to different CPUs (for more detail see Section 4.3).

2. The program cannot scale to simulate large populations due to the limited size of memory on the GPUs.

3. Portable memory allows sharing agent, household and business arrays among multiple GPUs.

4. Zero-copy may be performed, which is the best practice for the multiple GPU implementation.

In subsequent sections we highlight the parallelization of some critical elements of the simulation, such as pseudo-random number generator, agent initialization and assigning workplace and household to each agent, as well as the implementation of sending agents to locations. The entire pseudo-code can be found in the Appendix.
5.2.1 Pseudo-Random Number Generator

The agent-based model is a stochastic simulation. A pseudo-random number generator is used to assign workplaces for adults and schools for children, and generate daily schedules. Furthermore, the susceptible agents are chosen randomly by the infector to make contact with them at each location. We employ the cuRAND library to generate pseudo-random numbers for this simulation. The cuRAND library has tools that can generate efficient and high quality pseudo-random numbers. cuRAND is the only CUDA library that has APIs for the host side as well as the device side. The device API allows the developer to initialize and use pseudo-random numbers completely on the GPU. In this implementation, bit generation with XORWOW and MRG32k3a generators is utilized. A call to curand_init() initializes a sequence of pseudo-random numbers with a period greater than $2^{192}$. Functions curand() or curand_uniform() may be called to generate random numbers in another kernel. To avoid generation of the same sequence the curandState variable has to be updated [16].

5.2.2 Generating Agents

Implementing the Generate_Agents function (shown in Figure 5.1) in parallel is a challenge due to the enormous data dependency. In order to avoid data dependencies, we divide the sequential function into three steps. First, the population size is computed using a reduce function on the GPU, and three intermediate arrays are created, which store the number of adults, number of children as well as the total number of members in each candidate household (see Figure 5.3).
__global__ void compute_population_size(curandState *state,
unsigned long *Childsum,unsigned long *Adultsum,
float *household,int*Adult_intermed,int*Child_intermed,
unsigned char *dev_PPhouse)
{
__shared__ float cache[64];
__shared__ float cache2[64];
unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;
int j;
int temp=0;
int temp2=0;
curandState st=state[tid];
float n1=0;
for(j=0;j<houseperthread;j++)
{
    n1=curand_uniform(&st);
    int p=0;
    while(n1>household[p*3+2])
        p++;
    Adult_intermed[tid*houseperthread+j]=household[p*3+0];
    Child_intermed[tid*houseperthread+j]=household[p*3+1];
    dev_PPhouse[tid*houseperthread+j]=household[p*3+0] + household[p*3+1];
temp+=household[p*3+0];
    temp2+=household[p*3+1];
}
state[tid]=st;
cache[cacheIndex] = temp;
cache2[cacheIndex]= temp2;
__syncthreads();
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        {
            cache[cacheIndex] += cache[cacheIndex + i];
            cache2[cacheIndex]+=cache2[cacheIndex + i];
        }
    __syncthreads();
    i /= 2;
}
if (cacheIndex == 0)
{
    Childsum[blockIdx.x] = cache[0];
    Adultsum[blockIdx.x] = cache2[0];
}
}

Figure 5.3: Compute population size and generate intermediate arrays

Second, the adult and child intermediate arrays are copied form the device to the host. Write-combined pinned memory is used to allocate two arrays which are the adult agents and child agents. The purpose of allocating adult and child agent arrays is to store the household address for each agent that can further be accessed in parallel by kernels. In Figure 5.4 we illustrate the extraction of household
addresses from the adult intermediate array for adult agents. The index of the intermediate array represents a certain household and the content represents the number of adults in that household. The adult agent array is generated from the adult intermediate array by repeating the index of the adult intermediate array as many times as indicated by the number contained in the intermediate array cell. The household addresses for the child agent array is extracted in the same manner.

![Intermediate array diagram](image)

**Figure 5.4: Extracting array**

Third, the pointer of the adult agent array is passed to the Assign_Adult kernel (see Figure 5.5) to assign workplace, household as well as household members for each agent. In addition, the pointer of the child agent array and the size of adult array are passed to Assign_Child kernel to assign school, household, and household member. The adult agent size is used as a starting index for child agents in the overall agent array. As a result we get a presorted array of agents starting with adults and then followed by children.
__global__ void Assign_Adult(struct entity *Agents,unsigned long startpoint,
unsigned long POPULATION, curandState *state,
float *d_workplaces,unsigned char *dev_PPhouse,
unsigned long *Adult_agent)
{
    unsigned long tid=startpoint+threadIdx.x+blockIdx.x*blockDim.x;
    while(tid< POPULATION)
    {
        Agent[tid].id= tid;
        Agent[tid].household_number=Adult_agent[tid];
        Agent[tid].household_member=dev_PPhouse[ Adult_agent[tid] ];
        Agent[tid].workplace=assign_workplace();
        tid+=blockDim.x* gridDim.x;
    }
}

Figure 5.5: Assign_Adult kernel

5.2.3 Sending Agent to Locations

A schedule of 24 locations is generated, which represents each hour of the
day for all the agents in the simulation at the beginning of every simulated day.
However, in order to send agents to their scheduled location for the given hour,
the id of all present agents at that location has to be stored inside the location.
To implement this on GPU, we have chosen the atomicadd() function. The atomic
function performs a read-modify-write for each thread without interruption on
a shared memory structure. Threads are serialized for that part of the program.
Portable pinned memory is used in this simulation, which means atomic operations
on portable memory in one GPU are not atomic from the point of view of the CPU
or other GPUs in the system. As a consequence, the Send_AgentToLocation kernel
(depicted in Figure 5.6) has to be executed on one GPU and CudaDeviceSynchronize() function has to be called after launching Send_AgentToLocation() kernel,
to block CPU threads up until the kernel finishes its work.
```c
__global__ void Send_AgentToLocation(struct entity *Agent, struct bus *business,
                                       unsigned long size, int hour)
{
    unsigned long tid= threadIdx.x + blockIdx.x * blockDim.x;
    unsigned long bus;
    unsigned int temp;
    while(tid<size)
    {
        bus=Agent[tid].schedule[hour];
        if(bus>1 && business[bus].pointer<business_capacity)
        {
            temp=0;
            temp= atomicAdd(&business[bus].pointer,1);
            business[bus].count_total=temp+1;
            business[bus].array_id[temp]=community[tid].id;
        }
        tid+=blockDim.x * gridDim.x;
    }
}
```

**Figure 5.6: Send_AgentToLocations**

### 5.3 Multi-GPU

In general there are two approaches to call multiple GPUs. First, a single host can process multiple GPUs. The host thread calls a specific GPU to be current by calling `cudaSetDevice(int)`, then allocate the required memory for this device and launches one or more asynchronous kernels. Without waiting for the current GPU to become idle the host thread will set another device to be current. Second, the host is multi-threaded to access multiple GPUs, which is the approach we use our implementation. This approach creates one host thread per GPU.

Our implementation uses OpenMP for multi-threading. Each OpenMP thread calls `cudaSetDevice()` to set a specific GPU to be current. Since portable memory is used in the implementation, we cannot overlap kernels asynchronously. However, GPUs are executing the same kernel but the size of the portable memory is divided by the number of the GPUs; the first block is passed to the first GPU and the second block is passed to the second GPU, etc. The following pseudo-code in Figure 5.7 demonstrates the strategy of dividing pinned memory and launching the Schedule_Weekend() kernel on multiple GPUs.
```
numberOfGPUs = 4;
omp_set_num_threads(numberOfGPUs);
unsigned long start[numberOfGPUs], max[numberOfGPUs];
start[0]=0;
max[numberOfGPUs-1]= POPULATION;
for(int i = 1 ; i < numberOfGPUs ; i++)
{
    start[i] = (POPULATION/numberOfGPUs) * start[i-1];
    max[i-1] = start[i];
}
#pragma omp parallel
{
    int MyId=omp_get_thread_num();
cudaSetDevice(MyId);
cudaHostGetDevicePointer(&Agent[MyId],h_Agent,0);
Schedule_WeekEnd<<<Grid,Block>>>(Agent[MyId],d_state[MyId],start[MyId],max[MyId],
d_workplaces[MyId],d_index[MyId]);
if(cudaDeviceSynchronize()!= cudaSuccess)
    printf("fault at weekend GPU %d \n",MyId);
}
```

**Figure 5.7:** Splitting Agent into blocks and launching kernel on multiple GPUs
Chapter 6

Results and Conclusion

6.1 Results

The performance of the sequential implementation was measured for a population of up to 25 million individuals on a computer node of the "thor" cluster in Western Michigan University’s Parallel Computing and Data Science (PCDS) laboratory using dual Intel Xeon E5-2670 with 128 GB RAM. For a larger population size such as 50 and 100 million individuals, another computer node with Quad Intel Xeon E5-4640 and 512 GB is used for performance measurement due to the agent-based simulations large memory requirement. The performance of the multiple GPUs implementation is measured on thor’s node 7 that has four NVIDIA Tesla K20, 128 GB RAM and dual Intel Xeon E5-2670 processor.

For all executions, timing is performed using omp_get_wtime() function calls. Implemented simulations can simulate disease spreading for up to 120 days. However, for the purpose of runtime comparisons we run the simulation for 10 days due to the fact that the sequential execution for large numbers of days could take months to simulate. Moreover, the execution time for a population of 50 and 100 million individuals is estimated based on a one day simulation which
is the initialization duration time and one day loop duration time. The estimation is calculated by using $Init.time + (OneDay_{DurationTime} \times 10)$. For example, to simulate a population of 100 million individuals the initialization time is 314351.55s and one daily loop iteration time is 361821.132s, so the estimated time is $(314351.55 + (361821.132 \times 10))s$. The estimated sequential execution time are marked with an asterisk in the following table:

<table>
<thead>
<tr>
<th>Locations</th>
<th>Population</th>
<th>CPU Runtime</th>
<th>2GPUs Runtime</th>
<th>4GPUs Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1280</td>
<td>100000</td>
<td>26.91</td>
<td>1.3</td>
<td>0.95</td>
</tr>
<tr>
<td>12800</td>
<td>1000000</td>
<td>361.46</td>
<td>11.3</td>
<td>9.26</td>
</tr>
<tr>
<td>12800</td>
<td>2500000</td>
<td>1282.58</td>
<td>31.18</td>
<td>25.09</td>
</tr>
<tr>
<td>25600</td>
<td>5000000</td>
<td>4400.77</td>
<td>71.13</td>
<td>54.69</td>
</tr>
<tr>
<td>102400</td>
<td>10000000</td>
<td>18938.98</td>
<td>127.56</td>
<td>99.23</td>
</tr>
<tr>
<td>512000</td>
<td>25000000</td>
<td>745852.87</td>
<td>343.2</td>
<td>286.16</td>
</tr>
<tr>
<td>1024000</td>
<td>50000000</td>
<td>1612272.06*</td>
<td>711.37</td>
<td>553.89</td>
</tr>
<tr>
<td>1280000</td>
<td>100000000</td>
<td>3932562.87*</td>
<td>1473.87</td>
<td>1221.45</td>
</tr>
</tbody>
</table>

Table 6.1: Simulation Runtime Comparison in (sec)

![Runtime Comparison Figure](image-url)
Table 6.2 exhibits the speedup of two GPUs and four GPUs implementations relative to the sequential implementation. The speed up is calculated according to $Speedup = \frac{T_s}{T_p}$ where $T_s$ is the sequential CPU time and $T_p$ is the parallel time.

<table>
<thead>
<tr>
<th>Locations</th>
<th>Population</th>
<th>2GPUs Speedup</th>
<th>4GPUs Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1280</td>
<td>100000</td>
<td>20.7</td>
<td>28.32</td>
</tr>
<tr>
<td>12800</td>
<td>1000000</td>
<td>31.98</td>
<td>39.03</td>
</tr>
<tr>
<td>12800</td>
<td>2500000</td>
<td>41.13</td>
<td>51.11</td>
</tr>
<tr>
<td>25600</td>
<td>5000000</td>
<td>61.86</td>
<td>80.46</td>
</tr>
<tr>
<td>102400</td>
<td>10000000</td>
<td>148.47</td>
<td>190.85</td>
</tr>
<tr>
<td>512000</td>
<td>25000000</td>
<td>2173.23</td>
<td>2606.41</td>
</tr>
<tr>
<td>1024000</td>
<td>50000000</td>
<td>2266.43</td>
<td>2910.81</td>
</tr>
<tr>
<td>1280000</td>
<td>100000000</td>
<td>2668.18</td>
<td>3219.58</td>
</tr>
</tbody>
</table>

Table 6.2: Speedup

The population size is the most important component in the simulation. As the size of the population increases the performance of the sequential simulation decreases. The performance scales up with the population size enlargement for the multiple GPU implementation. However, the speedup levels off after the population reaches 25 million. This is due to several factors including memory swapping and data transmission over the QPI (QuickPath Interconnect) channel when GPU performs a read/write operations on the remote-memory (the memory that is connected to the other CPU in dual Sandy Bridge system, see 4.3).
If we compare the performance between the two GPUs and the four GPUs execution, we notice that even by doubling the number of GPUs we cannot double the performance on four GPUs due to memory overhead. Better performance can be achieved by decreasing memory usage and allocating the entire simulation objects on GPUs.

6.2 Future Work

We expect to improve the performance on 4 GPUs by:

- Splitting population in to four cities or States, to allocate the entire simulation objects (agent, location, household) on four GPUs and executing each city/State on one GPU. This requires model customization such as adding agent mobility between GPUs. For instance, an agent resident in the first GPU may run errands on the fourth GPU based on statistical data. We also want to perform peer-to-peer operations between GPUs that are connected to the same CPU. This allows streaming between GPUs without going through the CPU. Moreover, use a small window buffer on the host main
memory to transfer agents between GPUs where peer-to-peer operations are not allowed.

The following modifications are expected to increase the performance of all (1,2,4) GPU implementations:

- Replacing cuRAND by Random123. cuRAND requires 48 byte memory for each state and the state has to be updated after every curand() call function. In contrast, random123 is a Counter-Based Random Number Generator (CBRNG)[17]. It has a device-side API and it is stateless, meaning that it does not require memory to generate pseudo-random numbers.

- Sorting agents by their presence at locations in scheduled hours and storing the first agent and last agent indices at the location array. Further, agents can make contacts at their current location within the stored range based on the contact rate. Using this approach, only 9.77MB of memory is needed for location array of size 1.28 million which is used when simulating a population of size 100 million individuals.

6.3 Conclusion

In this thesis, we demonstrated a high-performance framework of a low-complexity agent-based model for pandemic simulation. The contact network of our agent-based simulation is able to include actual data from the field based social contact. The epidemiological model can accept observational field data approximation of the initial reproduction number, and then use it to calibrate the infection probability. This is done in such a way that the results of the infection pattern is similar to the value of the basic reproduction number.

Our multiple GPUs implementation exhibits remarkable improvement in execution runtime which permits real time processing. For example, FluTE simulation
requires two hours on a cluster environment to simulate a population of size ten million individuals [8]. In contrast, our simulation on a single machine accelerated using multiple GPUs can simulate 100 days in 89.6 seconds for a population of size ten million individuals.

This thesis provides a fundamental framework for implementation of HPC simulations with GPU acceleration. We achieved speedups of up to 2606.41, which is relatively faster than the 3.3x achieved by Barrett et al. [18], or the 11.7x by Zou et al. [19] or the 94.4x achieved by Holvenstot et al. [4]. Moreover, we achieved a substantial simulation scaling of up to 100 million individuals using multiple GPUs.
Appendix A

An Appendix

A.1 GPU Kernels Source Code

```c
__global__ void compute_population_size(curandState *state,
    unsigned long *c,
    unsigned long *d,
    float *household,
    int*dev_adult,
    int*dev_child,
    unsigned char *dev_PPhouse)
{
    __shared__ float cache[64];
    __shared__ float cache2[64];
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    int j;
    int temp=0;
    int temp2=0;
    curandState st=state[tid];
    float n1=0;
    for(j=0;j<houseperthread;j++)
    {
        n1=curand_uniform(&st);
        int p=0;
        while(n1>household[p*3+2])
            p++;
    }
```
dev_adult[tid*houseperthread+j]=household[p*3+0];
dev_child[tid*houseperthread+j]=household[p*3+1];
dev_PPhouse[tid*houseperthread+j]=household[p*3+0] + household[p*3+1];
temp+=household[p*3+0];
temp2+=household[p*3+1];
}
state[tid]=st;
cache[cacheIndex] = temp;
cache2[cacheIndex]= temp2;
__syncthreads();
int i = blockDim.x/2;
while (i != 0)
{
    if (cacheIndex < i)
    {
        cache[cacheIndex] += cache[cacheIndex + i];
        cache2[cacheIndex]+=cache2[cacheIndex + i];
    }
    __syncthreads();
    i /= 2;
}
if (cacheIndex == 0)
{
    c[blockIdx.x] = cache[0];
    d[blockIdx.x] = cache2[0];
}

__global__ void init_rand(curandState *state, int seed)
{
    unsigned long idx=threadIdx.x+blockIdx.x* blockDim.x;
    curand_init(seed,idx,0,&state[idx]);
}

__global__ void generate_businesses(unsigned long *index,
unsigned long SIZE,
float *d_workplaces,
unsigned char * contact_rate,
int v) // generate all business entities
{
    __shared__ float lworkplaces[14][7];
    long i,j;
    for(i=0;i<14;i++)
for(j=0;j<7;j++)
{
    lworkplaces[i][j]=d_workplaces[i*7+j];
}

unsigned long idx=threadIdx.x+blockIdx.x*blockDim.x;
while(idx<SIZE)
{
    if(idx<index[0])
    {
        contact_rate[idx] = lworkplaces[0][6];
    }
    else if(idx>=index[0] && idx<index[1])
    {
        contact_rate[idx] = lworkplaces[1][6];
    }
    else if(idx>=index[1] && idx<index[2])
    {
        contact_rate[idx] = lworkplaces[2][6];
    }
    else if(idx>=index[2] && idx<index[3])
    {
        contact_rate[idx] = lworkplaces[3][6];
    }
    else if(idx>=index[3] && idx<index[4])
    {
        contact_rate[idx] = lworkplaces[4][6];
    }
    else if(idx>=index[4] && idx<index[5])
    {
        contact_rate[idx] = lworkplaces[5][6];
    }
    else if(idx>=index[5] && idx<index[6])
    {
        contact_rate[idx] = lworkplaces[6][6];
    }
    else if(idx>=index[6] && idx<index[7])
    {
        contact_rate[idx] = lworkplaces[7][6];
    }
    else if(idx>=index[7] && idx<index[8])
    {
        contact_rate[idx] = lworkplaces[8][6];
    }
    else if(idx>=index[8] && idx<index[9])
    {


contact_rate[idx] = lworkplaces[9][6];
}
else if(idx>=index[9] && idx<index[10])
{
  contact_rate[idx] = lworkplaces[10][6];
}
else if(idx>=index[10] && idx<index[11])
{
  contact_rate[idx] = lworkplaces[11][6];
}
else if(idx>=index[11] && idx<index[12])
{
  contact_rate[idx] = lworkplaces[12][6];
}
else if(idx>=index[12])
{
  contact_rate[idx] = lworkplaces[13][6];
}
idx+=blockDim.x*gridDim.x;


__device__ unsigned long Assignworkplace(float *d_workplaces, curandState state)
{
  float sum=0.PRNG;
  PRNG=curand_uniform(&state);
  unsigned long work=0;
  for(int i=0;i<14;i++)
  {
    sum+=d_workplaces[i*7+2];
    if(sum>PRNG)
    {
      sum-=PRNG;
      sum=sum/d_workplaces[i*7+2];
    sum=sum* d_workplaces[i*7+1];
      return (unsigned long) work+sum;
    }
    else
      work+= (unsigned int) d_workplaces[i*7+1];
  }
  return work;
}
__global__ void Assign_Adult(struct entity *community,
unsigned long startpoint,
unsigned long POPULATION,
curandState *state,
float *d_workplaces,
unsigned char *dev_PPhouse,
unsigned long *dev_adultset)
{
    unsigned long tid=startpoint+threadIdx.x+blockIdx.x*blockDim.x;
    int i=0;
    unsigned long workpla=0;
    curandState st;
    while(tid< POPULATION)
    {
        community[tid].id= tid;
        community[tid].household_number=dev_adultset[tid];
        community[tid].household_member=dev_PPhouse[ dev_adultset[tid] ];
        community[tid].infection_day_pandemic = -6;
        community[tid].infection_day_seasonal = -6;
        st=state[(tid%(128*128))];
        community[tid].workplace = Assignworkplace(d_workplaces,st);
        state[(tid%(128*128))]=st;
        community[tid].age=23;
        tid+=blockDim.x* gridDim.x;
    }
}

__global__ void Assign_Child(struct entity *community,
unsigned long POPULATION,
unsigned long Adult,
unsigned long startpoint,
curandState *state,
unsigned char *dev_PPhouse,
unsigned long *dev_childset,
float *dev_AgeChild,
unsigned long *index)
{
__shared__ float age[5][4];
    unsigned long tid=startpoint+threadIdx.x+blockIdx.x*blockDim.x;
    unsigned long ctid=0;
    int i=0,j=0;
curandState st;
float PRNG;
for(int i=0;i<5;i++)
    for(int j=0;j<3;j++)
        age[i][j]=dev_AgeChild[i*3+j];
while(tid < POPULATION)
{
    ctid=tid+Adult;
    community[ctid].id= ctid;
    community[ctid].household_number=dev_childset[tid];
    community[ctid].household_member=dev_PPhouse[ dev_childset[tid] ];
    community[ctid].infection_day_pandemic = -6;
    community[ctid].infection_day_seasonal = -6;
    st=state[(tid%(128*128))];
    PRNG=curand_uniform(&st);
    j=0;
    while(PRNG>age[j][1]&& j<5)
    {
        j++;
        community[ctid].age=dev_AgeChild[j*3];
        long randval=curand(&st);
        long value=0;
        if(j!=0)
        {
            value= randval % (1+index[3+j]-index[3+(j-1)]);
            value+=index[3+(j-1)];
        }
        else
        {
            value= randval %(1+index[3]-index[2]);
            value+=index[2];
        }
        community[ctid].workplace = value;
        state[(tid%(128*128))] = st;
        tid+=blockDim.x* gridDim.x;
    }
}
__global__ void initialpandemic(int size, curandState *state,
                                unsigned long POPULATION,
                                float reproduction,
                                struct entity *community)
{
    int tid=threadIdx.x + blockIdx.x* blockDim.x;
    unsigned long index,person;
    if(tid<size)
curandState st=state[tid];
index=curand(&st);
person= index %(1+POPULATION);
community[person].disease_clock_pandemic=0;
community[person].generation_pandemic=1;
community[person].infection_day_pandemic=-2;
community[person].virus=1;
community[person].prior_virus=1;
community[person].rn_init_pandemic=reproduction;
state[tid]=st;
}

__global__ void initialseasonal(int size, curandState *state,
unsigned long POPULATION,
float reproduction,
struct entity *community)
{
int tid=threadIdx.x + blockIdx.x* blockDim.x;
unsigned long index,person;
if(tid<size)
{
curandState st=state[tid];
index=curand(&st);
person= index %(1+POPULATION);
community[person].disease_clock_seasonal=0;
community[person].generation_seasonal=1;
community[person].infection_day_seasonal=-2;
community[person].virus=2;
community[person].prior_virus=2;
community[person].rn_init_seasonal=reproduction;
state[tid]=st;
}
}

__global__ void ChildWeekDay(unsigned long startpoint,unsigned long size,
unsigned long adult, struct entity *community,
unsigned long *index, float *d_workplaces, curandState *state)
{
unsigned long tid=startpoint+threadIdx.x*blockIdx.x*blockDim.x;
unsigned long ctid=0;
int i=0;
long PRNG=0,work=0;
curandState st;
while(tid<size)
{

tid=tid+adult;
for(i=0;i<24;i++)
{
    if(i>6 && i<15)
        community[tid].schedule[i]=community[tid].workplace;
    else
        community[tid].schedule[i]=1;
}
st=state[tid%(128*128)];
PRNG=curand(&st);
work=PRNG%(index[8]+1-index[7]);
community[tid].schedule[17]=index[7]*work;
PRNG=curand(&st);
work=PRNG%(index[8]+1-index[7]);
community[tid].schedule[18]=index[7]*work;
state[tid%(128*128)]=st;
tid+=blockDim.x*gridDim.x;
}

__global__ void AdultWeekDay(unsigned int startpoint,unsigned long size,
struct entity *community, unsigned long *index,
float *d_workplaces, curandState *state )
{
    unsigned long tid=startpoint+threadIdx.x*blockIdx.x*blockDim.x;
    int i=0;
    unsigned long value=0;
    float PRNG=0;
    curandState st;
    while(tid<size)
    {
        for(i=0;i<24;i++)
        {
            if(i>6 && i<17)
                community[tid].schedule[i]=community[tid].workplace;
            else
                community[tid].schedule[i]= 1;
        }
        st=state[tid%(128*128)];
        for(int j=1;j<3;j++)
        {
            float sum=0;
            PRNG=curand_uniform(&st);
        }
for(i=9;i<

} 
number_business_type;i++)
{
    sum = sum + d_workplaces[i+7]+3;
    if(PRNG< sum ||i==number_business_type-1)
    {
        sum-=PRNG;
        sum=sum/d_workplaces[i+7+3];
        sum==d_workplaces[i+7+1];
        value=index[i+1]+(long)sum;
        break;
    }
}

community[tid].schedule[i6+j]=value;
}
state[(tid%(128*128))]=st;
tid=blockDim.x *gridDim.x;
}

__global__ void WeekEnd(struct entity *community, curandState *state,
unsigned long start, unsigned long size,
float *d_workplaces, unsigned long *index)
{
    unsigned long value, tid=start+threadIdx.x+ blockIdx.x*blockDim.x;
    unsigned long a[3], swap=0, i=0;
    curandState st;
    float PRNG=0;
    long rnd=0;
    while(tid < size)
    {
        for(i=0;i<24;i++)
            community[tid].schedule[i]=1;
        st=state[tid%(128*128)];
        rnd= curand(&st);
        rnd = (rnd %6);
        a[0]=curand(&st); a[0]= (a[0] % 11) +9;
        a[1]=curand(&st); a[1]= (a[1] % 11) +9;
        while(a[0]==a[1])
        {
            a[1]=curand(&st); a[1]= (a[1] % 11)+9;
        }
        if(a[1]<a[0])
        {
            swap=a[0]; a[0]=a[1]; a[1]=swap;
a[2] = curand(&st);
while (a[0] == a[2] || a[1] == a[2]) {
}
if (a[2] < a[0]) {
    swap = a[2]; a[2] = a[1]; a[1] = a[0]; a[0] = swap;
} else {
    if (a[2] < a[1]) {
        swap = a[2]; a[2] = a[1]; a[1] = swap;
    }
}
for (int j = 0; j < 3; j++) {
    float sum = 0;
    PRNG = curand_uniform(&st);
    for (i = 9; i < number_business_type; i++) {
        sum += d_workplaces[i*7 + 4];
        if (PRNG < sum || i == number_business_type - 1) {
            sum -= PRNG;
            sum /= d_workplaces[i*7 + 4];
            sum *= d_workplaces[i*7 + 1];
            value = index[i - 1] + (long)sum;
            break;
        }
    }
    community[tid].schedule[a[j]] = value;
    community[tid].errandw[j] = a[j];
}
state[(tid % (128 * 128))] = st;
tid += blockDim.x * gridDim.x;

__global__ void DiseaseProgPandemic(struct entity *community, int* PandemicReduce,
                                      unsigned long size, int hour)
unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
while (tid < size)
{
    if (community[tid].infection_day_pandemic == -6)
        community[tid].disease_clock_pandemic = 0;

    if (community[tid].disease_clock_pandemic >= 0 &&
        community[tid].disease_clock_pandemic < culmination_period)
    {
        if (community[tid].infection_day_pandemic > 0)
            **community[tid].disease_clock_pandemic;
        community[tid].disease_clock_pandemic;
        community[tid].infection_day_pandemic = (int)(community[tid].disease_clock_pandemic/13)+1;
    }
    else{
        if (community[tid].infection_day_pandemic == -2 && hour == 7)
            **community[tid].disease_clock_pandemic;
        community[tid].disease_clock_pandemic;
        community[tid].infection_day_pandemic = (int)(community[tid].disease_clock_pandemic/13)+1;
    }
}

if (community[tid].disease_clock_pandemic >= culmination_period)
{
    community[tid].disease_clock_pandemic = -3;
    community[tid].infection_day_pandemic = -5;
    PandemicReduce[tid] += 1;
}

tid += blockDim.x * gridDim.x;
}

__global__ void DiseaseProgSeasonal(struct entity *community, int *SeasonalReduce,
unsigned long size, int hour)
{
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
while (tid < size)
{
    if (community[tid].infection_day_seasonal == -6)
    {
        if (community[tid].infection_day_seasonal == -6)
            community[tid].disease_clock_seasonal = 0;
    }
}

// Additional code for DiseaseProgSeasonal
community[tid].disease_clock_seasonal=0;

if(community[tid].disease_clock_seasonal >=0 &&
   community[tid].disease_clock_seasonal < culmination_period)
{
  if(community[tid].infection_day_seasonal >0)
  {
    ++community[tid].disease_clock_seasonal;
    community[tid].infection_day_seasonal= (int)(community[tid].disease_clock_seasonal/13)+1;
  }
  else{
    if(community[tid].infection_day_seasonal== -2 && hour ==7)
    {
      ++community[tid].disease_clock_seasonal;
      community[tid].infection_day_seasonal= (int)(community[tid].disease_clock_seasonal/13)+1;
    }
  }
}
else{
  if(community[tid].disease_clock_seasonal >= culmination_period)
  {
    community[tid].disease_clock_seasonal = -3;
    community[tid].infection_day_seasonal = -5;
    SeasonalReduce[tid]**=1;
  }
}
tid+=blockDim.x*gridDim.x;
}

__global__ void ReduceReproduction(struct entity *community,unsigned long *SemiSea,
unsigned long *SemiSea2, unsigned long *Semisum,
unsigned long *Semisum2,unsigned long size,
unsigned long startpoint,int day)
{
  __shared__ unsigned int temp[64],bin[64];
  unsigned long tid= startpoint+threadIdx.x + blockIdx.x* blockDim.x;
  int thrd=threadIdx.x;
  unsigned long value=0,rn_p=0, generation=0;
  while(tid< size)
  {
    generation=community[tid].generation_pandemic;
    if(generation==day)

value+=1;
    rn_p+=community[tid].rn_pandemic;
}
tid+=blockDim.x*gridDim.x;
}
temp[thrd]=value;
bin[thrd]=rn_p;
__syncthreads();
int i=blockDim.x/2;
while(i!=0)
{
    if(thrd<i)
    {
        temp[thrd]+=temp[thrd+i];
        bin[thrd]+=bin[thrd+i];
    }
    __syncthreads();
    i/=2;
}
if(thrd==0)
{
    Semisum[blockIdx.x]=temp[0];
    Semisum2[blockIdx.x]=bin[0];
}
}

__global__ void init_households(struct house *household,unsigned long size,unsigned long startpoint)
{
    unsigned long tid=startpoint+threadIdx.x+ blockIdx.x * blockDim.x;
    unsigned int i=0,max=0;
    while(tid<size)
    {
        max=household[tid].count_total;
        household[tid].pointer=0;
        for(i=0;i<max;i++)
            household[tid].array_id[i]=0;
        household[tid].count_total=0;
        tid+=blockDim.x * gridDim.x;
    }
}

__global__ void init_business(struct bus *business,unsigned long size,unsigned long idx)
{  
    unsigned long tid = idx + threadIdx.x * blockDim.x * gridDim.x;  
    unsigned int i = 0, max = 0;  
    while (tid < size)  
    {  
        max = business[tid].count_total;  
        business[tid].count_infected = 0;  
        business[tid].pointer = 0;  
        for (i = 0; i < max; i++)  
            business[tid].array_id[i] = 0;  
        business[tid].count_total = 0;  
        tid += blockDim.x * gridDim.x;  
    }  
}

__global__ void declare_locationBusiness(struct entity *community, struct bus *business, 
                                        unsigned long size, int hour)  
{  
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;  
    unsigned long bus;  
    unsigned int temp;  
    while (tid < size)  
    {  
        bus = community[tid].schedule[hour];  
        if (bus > 1 && business[bus].pointer < business_capacity)  
        {  
            temp = 0;  
            temp = atomicAdd(&business[bus].pointer, 1);  
            business[bus].count_total = temp + 1;  
            business[bus].array_id[temp] = community[tid].id;  
        }  
        tid += blockDim.x * gridDim.x;  
    }  
}

__global__ void declarehouse(struct entity *community, struct house *household, 
                              unsigned long size, int hour)  
{  
    unsigned long tid = blockIdx.x * blockDim.x + threadIdx.x;  
    unsigned long bus = 0, house = 0, id;  
    unsigned int temp, member = 0;  
    while (tid < size)  
    {  

bus=community[tid].schedule[hour];
house=community[tid].household_number;
member=community[tid].household_member;
id=community[tid].id;
if(bus==1 && household[house].pointer<member)
{
    temp=0;
    temp=atomicAdd(&household[house].pointer,1);
    household[house].count_total=temp+1;
    household[house].array_id[temp]=id;//community[tid].id;
}
tid+=blockDim.x*gridDim.x;
}

__global__ void symptomaticpandemic(struct entity *community, curandState *state, unsigned long start, unsigned long size, int hour)
{
    unsigned long tid=start+ threadIdx.x + blockIdx.x * gridDim.x;
curandState st;
float PRNG;
unsigned int value;
while(tid < size)
{
    st=state[(tid%((128*128)))];
    if (community[tid].disease_clock_pandemic >= 1 &&
        community[tid].disease_clock_pandemic < culmination_period)
    {
        if( community[tid].infection_day_pandemic>0 &&
            community[tid].symptomatic==0 && community[tid].virus==1) // for the individuals initiating the outbreak
            PRNG=0;
            value=0;
            PRNG=curand_uniform(&st);
            if( PRNG<= percent_symptomatic)
            {
                community[tid].symptomatic=1;
                value=curand(&st); value%=3;
                if(value==0)
                    value=3;
                community[tid].profile_pandemic= value ;
            }
    }
    else
    {
}
community[tid].symptomatic=2;
value=curand(&st);
community[tid].profile_pandemic = (value%3)+4;
}
}
}
}
}
state[(tid%(128*128))]=st;
tid+= blockDim.x * gridDim.x;

__global__ void symptomaticseasonal(struct entity *community,curandState *state,
unsigned long start, unsigned long size,int hour)
{
unsigned long tid=start+threadIdx.x + blockIdx.x * blockDim.x;
curandState st;
float PRNG;
unsigned int value;
while(tid <size)
{
    st=state[(tid%(128*128))];
    if (community[tid].disease_clock_seasonal >= 1 &&
        community[tid].disease_clock_seasonal < culmination_period &&
        community[tid].infection_day_seasonal>0 &&community[tid].symptomatic==0 &&
        community[tid].virus==2)
    {
        PRNG=0;
        value=0;
        PRNG=curand_uniform(&st);
        if(PRNG <= percent_symptomatic)
        {
            community[tid].symptomatic=1;
            value = curand(&st); value%=3;
            if(value==0)
                value=3;
            community[tid].profile_seasonal=value;
        }
        else
        {
            community[tid].symptomatic=2;
            value=curand(&st);
            community[tid].profile_seasonal=(value %3)+4;
        }
    }
}
__global__ void AdultdiseaseSpreadWday(struct entity *community, struct bus *business,
unsigned long size, int hour, curandState *state,
unsigned char* contact_rate)
{
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned long location;
    int contactRate, Rate, count;
    curandState st;
    while(tid < size)
    {
        location=community[tid].schedule[hour];
        count= business[location].count_total;
        st=state[(tid%(128*128))];
        if(hour==7)
        {
            contactRate= contact_rate[location];
            if(business[location].count_total <= contactRate)
            {
                contactRate=business[location].count_total-1;
            }
            if(contactRate > 0)
            {
                for (int i = 0; i < contactRate; i++)
                {
                    Rate = curand(&st); Rate = Rate % count;
                    while(business[location].array_id[Rate] == community[tid].id)
                    {
                        Rate = curand(&st); Rate = Rate % count;
                    }
                    community[tid].contact_array_id[community[tid].contact_counter+i]=
                    business[location].array_id[Rate];
                }
                community[tid].contact_counter += contactRate;
                community[tid].contact_counter_wp +=contactRate;
            }
        }
        if(hour==17)
        {
            contactRate= contact_rate[location];
        }
    }
}
if (business[location].count_total <= contactRate)
{
    contactRate = business[location].count_total - 1;
}
if (contactRate > 0)
{
    for (int i = 0; i < contactRate; i++)
    {
        Rate = curand(&st);
        Rate = Rate % count;
        while (business[location].array_id[Rate] == community[tid].id)
        {
            Rate = curand(&st); Rate = Rate % count;
        }
        community[tid].contact_array_id[community[tid].contact_counter + i] =
        business[location].array_id[Rate];
    }
    community[tid].contact_counter += contactRate;
    community[tid].contact_counter_er += contactRate;
}
state[(tid % (128 * 128))] = st;
tid += blockDim.x * gridDim.x;

__global__ void ChilddiseaseSpreadWday(struct entity *community, struct bus *business,
unsigned long size, unsigned long adult, int hour,
curandState *state, unsigned char *contact_rate)
{
    unsigned long tid = adult + threadIdx.x + blockIdx.x * blockDim.x;
    unsigned long location;
    int contactRate, Rate, count;
    curandState st;
    while (tid < size)
    {
        location = community[tid].schedule[hour];
        count = business[location].count_total;
        st = state[(tid % (128 * 128))];
        if (hour == 7)
        {
            contactRate = contact_rate[location];
            if (business[location].count_total <= contactRate)
            {
                

59
contactRate = business[location].count_total - 1;
}
if (contactRate > 0)
{
    for (int i = 0; i < contactRate; i++)
    {
        Rate = curand(&st); Rate = Rate % count;
        while (business[location].array_id[Rate] == community[tid].id)
        {
            Rate = curand(&st); Rate = Rate % count;
        }
        community[tid].contact_array_id[community[tid].contact_counter+i] =
            business[location].array_id[Rate];
    }
    community[tid].contact_counter += contactRate;
    community[tid].contact_counter_wp = contactRate;
}
if (hour == 15)
{
    contactRate = contact_rate[location];
    if (business[location].count_total <= contactRate)
    {
        contactRate = business[location].count_total - 1;
    }
    if (contactRate > 0)
    {
        for (int i = 0; i < contactRate; i++)
        {
            Rate = curand(&st); Rate = Rate % count;
            while (business[location].array_id[Rate] == community[tid].id)
            {
                Rate = curand(&st);
                Rate = Rate % count;
            }
            community[tid].contact_array_id[community[tid].contact_counter+i] =
                business[location].array_id[Rate];
        }
        community[tid].contact_counter += contactRate;
        community[tid].contact_counter_er = contactRate;
    }
}
state[(tid%(128*128))] = st;
tid += blockDim.x * gridDim.x;
__global__ void housediseasespreadWd(struct entity *community,
  struct house *household,
  unsigned long size,
  curandState *state, int hour)
{
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
    int contactRate;
    unsigned long Rate;
    unsigned long house, count;
    curandState st;
    while (tid < size)
    {
      house = community[tid].household_number;
      count = household[house].count_total;
      st = state[tid%(128*128)];
      contactRate = 3;
      if (household[house].count_total <= contactRate)
        contactRate = household[house].count_total - 1;
      if (contactRate > 1)
      {
        for (int i = 0; i < contactRate; i++)
        {
          Rate = curand(&st); Rate = Rate % count;
          while (household[house].array_id[Rate] == community[tid].id)
          {
            Rate = curand(&st);
            Rate = Rate % count;
          }
          community[tid].contact_array_id[community[tid].contact_counter+i] =
            household[house].array_id[Rate];
        }
        community[tid].contact_counter += contactRate;
        community[tid].contact_counter_hh = contactRate;
      }
      state[tid%(128*128)] = st;
      tid += blockDim.x * gridDim.x;
    }
  }
__global__ void d_infection(struct entity *community, curandState *state,
    unsigned long size, float reproduction_number_pandemic,
    float reproduction_number_seasonal, int day,
    float *d_gamma1, float *d_gamma2, float *d_lognorm1,
    float *d_lognorm2, float *d_weib1, float *d_weib2)
{
    unsigned long tid = threadIdx.x + blockIdx.x * blockDim.x;
    int virus, k=0;
    curandState st;
    float p, z, p_p, z_p, z_s, PRNG;
    unsigned long id, value;
    while(tid < size)
    {
        st = state[(tid%(128*128))];
        virus = community[tid].virus;
        for(int i=0;i<community[tid].contact_counter;i++)
        {
            id = community[tid].contact_array_id[i]; k=1;
            p = -1;
            z = -1;
            if(virus == 3)
            {
                if(community[tid].infection_day_pandemic == -2)
                {
                    p_p = 0;
                }
                else
                {
                    float a = 0, repnum = community[tid].rn_init_pandemic;
                    int profile = community[tid].profile_pandemic;
                    int infection_day = community[tid].infection_day_pandemic;
                    switch (profile)
                    {
                        case 1:
                            a = d_gamma1[infection_day]; break;
                        case 2:
                            a = d_lognorm1[infection_day]; break;
                        case 3:
                            a = d_weib1[infection_day]; break;
                        case 4:
                            a = d_gamma2[infection_day]; break;
                        case 5:
                            a = d_lognorm2[infection_day]; break;
                        case 6:
                            a = d_weib2[infection_day]; break;
                    }
                    p_p = a;
                }
            }
            else
            {
                p_p = p;
            }
        }
    }
}
if(a!=0)
    p_p = ((repnum/(((1-asympt)+percent_symptomatic)+asympt))*a)/
        ((community[tid].contact_counter_hh*k_hh)+
        (community[tid].contact_counter_wp*k_wp)+
        (community[tid].contact_counter_er*k_er));
    }
    z_p=curand_uniform(&st);
    if(community[tid].infection_day_seasonal==-2)
    {
        p_s=0;
    }
    else
    {
        float a=0,repnum= community[tid].rn_init_seasonal;
        int profile= community[tid].profile_seasonal;
        int infection_day=community[tid].infection_day_seasonal;
        switch (profile)
        {
            case 1:
                a = d_gamma1[infection_day]; break;
            case 2:
                a = d_lognorm1[infection_day]; break;
            case 3:
                a = d_weib1[infection_day]; break;
            case 4:
                a = d_gamma2[infection_day]; break;
            case 5:
                a = d_lognorm2[infection_day]; break;
            case 6:
                a = d_weib2[infection_day]; break;
        }
        if(a!=0)
            p_s = ((repnum/(((1-asympt)+percent_symptomatic)+asympt))*a)/
                ((community[tid].contact_counter_hh*k_hh)+
                (community[tid].contact_counter_wp*k_wp)+
                (community[tid].contact_counter_er*k_er));
    }
    z_s=curand_uniform(&st);
    if (z_p <= p_p && z_s > p_s)
    {//infected from pandemic
        virus = 1; p = p_p; z = z_p;
    }
    if (z_s <= p_s && z_p > p_p)
    {//infected from seasonal

virus = 2; p = p_s; z = z_s;
}
if (z_p <= p_p && z_s <= p_s)
{
    if (community[id].prior_virus==0)
    {//no prior infection, infected with pandemic and seasonal
        PRNG=curand_uniform(&st);
        if(PRNG <= percent_symptomatic)
        {
            community[id].symptomatic=1;
            value= curand(&st); value%=3;
            if(value==0)
                value=3;
            community[id].profile_pandemic=value;
        }
        else
        {
            community[id].symptomatic=2;
            value=curand(&st);
            community[id].profile_pandemic=(value %3)+4;
        }
        PRNG=curand_uniform(&st);
        if(PRNG <= percent_symptomatic)
        {
            community[id].symptomatic=1;
            value = curand(&st);
            value%=3;
            if(value==0)
                value=3;
            community[id].profile_seasonal=value ;
        }
        else
        {
            community[id].symptomatic=2;
            value=curand(&st);
            community[id].profile_seasonal=(value%3)+4;
        }
        community[id].disease_clock_pandemic = 0;
        community[id].disease_clock_seasonal = 0;
        community[id].generation_pandemic = community[tid].generation_pandemic + 1;
        community[id].generation_seasonal = community[tid].generation_seasonal + 1;
        community[tid].rn_pandemic = community[tid].rn_pandemic + 1;
        community[tid].rn_seasonal = community[tid].rn_seasonal + 1;
        community[id].infection_day_pandemic = -2;
        community[id].infection_day_seasonal = -2;
    }
if (community[id].prior_virus==2 && community[id].disease_clock_seasonal>=0)
{//Infected with seasonal,
//still in the seasonal process and infected with both pandemic and seasonal
virus = 1;
p = p_p;
z = z_p;
}
if (community[id].prior_virus==2 && community[id].disease_clock_seasonal==-3 &&
    community[id].infection_day_seasonal==-5)
{
    virus = 1;//The person will get infected from pandemic only
    p = p_p;
z = z_p;
}
if (community[id].prior_virus==1 && community[id].disease_clock_pandemic>=0)
{
    virus = 2;//The person will get infected from seasonal only
    p = p_s; z = z_s;
}
if (community[id].prior_virus==1 && community[id].disease_clock_pandemic==-3 &&
    community[id].infection_day_pandemic==-5)
{
    virus = 2;//The person will get infected from seasonal only
    p = p_s; z = z_s;
}
} //end of virus 3
```c
else
{
    float a=0,repnum= community[tid].rn_init_pandemic;
    int profile= community[tid].profile_pandemic;
    int infection_day=community[tid].infection_day_pandemic;
    switch (profile)
    {
    case 1:
        a = d_gamma1[infection_day]; break;
    case 2:
        a = d_lognorm1[infection_day]; break;
    case 3:
        a = d_weib1[infection_day]; break;
    case 4:
        a = d_gamma2[infection_day]; break;
    case 5:
        a = d_lognorm2[infection_day]; break;
    case 6:
        a = d_weib2[infection_day]; break;
    }
    if(a!=0)
    p = (((repnum/((1-asymp)*percent_symptomatic)+asymp))*a)/
        ((community[tid].contact_counter_hh*k_hh)+
        (community[tid].contact_counter_wp*k_wp)+
        (community[tid].contact_counter_er*k_er));
}
}
if (z==-1)
{
    z=curand_uniform(&st);// uni(0,1);
}
if (z <= (k*p)|| p==0)
{
    if (community[id].prior_virus==0)
    {//no prior infection, infected with pandemic
        PRNG=curand_uniform(&st);
        if(PRNG <= percent_symptomatic)
        {
            community[id].symptomatic=1;
            value= curand(&st); value%=3;
            if(value==0)
            value=3;
            community[id].profile_pandemic=value;
        }
    }
else
```
{  
    community[id].symptomatic=2;  
    value=curand(&st);  
    community[id].profile_pandemic=(value %3)+4;  
}  
community[id].disease_clock_pandemic = 0;  
community[id].generation_pandemic = community[tid].generation_pandemic + 1;  
community[tid].rn_pandemic = community[tid].rn_pandemic + 1;  
community[id].infection_day_pandemic = -2;  
community[id].day_begin_infection_pandemic = day;  
community[id].virus = 1;  
community[id].prior_virus = 1;  
community[id].rn_init_pandemic = reproduction_number_pandemic;  
community[id].total_infected_pandemic++=1;  
community[id].total_infected_pandemic_only++=1;  
}  
if (community[id].prior_virus==2 && community[id].disease_clock_seasonal>=0)  
  //Infected with seasonal, still in the seasonal process and infected with pandemic  
  PRNG=curand_uniform(&st);  
  if(PRNG <= percent_symptomatic)  
    {  
      community[id].symptomatic=1;  
      value= curand(&st);  
      value%=3;  
      if(value==0)  
        value=3;  
      community[id].profile_pandemic=value ;  
    }  
  else  
    {  
      community[id].symptomatic=2;  
      value=curand(&st);  
      community[id].profile_pandemic=(value %3)+4;  
    }  
  community[id].disease_clock_pandemic = 0;  
  community[id].generation_pandemic = community[tid].generation_pandemic + 1;  
  community[tid].rn_pandemic = community[tid].rn_pandemic + 1;  
  community[id].infection_day_pandemic = -2;  
  community[id].day_begin_infection_pandemic = day;  
  community[id].virus = 3;  
  community[id].prior_virus = 3;  
  community[id].rn_init_pandemic = reproduction_number_pandemic*epsilon_p;  
  community[id].total_infected_pandemic++=1;  
  community[id].total_coinfected_s_p++=1;  
}
if (community[id].prior_virus==2 && community[id].disease_clock_seasonal==-3 &&
community[id].infection_day_seasonal==-5)
{//Infected with seasonal, recovered from seasonal process and infected with pandemic
PRNG=curand_uniform(&st);
if(PRNG <= percent_symptomatic)
{
    community[id].symptomatic=1;
    value= curand(&st);
    value%=3;
    if(value==0)
    {
        value=3;
        community[id].profile_pandemic=value ;
    }
    else
    {
        community[id].symptomatic=2;
        value=curand(&st);
        community[id].profile_pandemic=(value %3)+4;
    }
    community[id].disease_clock_pandemic = 0;
    community[id].generation_pandemic = community[tid].generation_pandemic + 1;
    community[tid].rn_pandemic = community[tid].rn_pandemic + 1;
    community[id].infection_day_pandemic = -2;
    community[id].day_begin_infection_pandemic = day;
    community[id].virus = 1;
    community[id].prior_virus = 3;
    community[id].rn_init_pandemic = reproduction_number_pandemic*epsilon_pr;
    community[id].total_infected_pandemic+=1;
}
}else
{// end of if (z <= (k*p))
//end of virus 1
if(virus==2)
{
    if (p==1)
    {
        float a=0,repnum= community[tid].rn_init_seasonal;
        int profile= community[tid].profile_seasonal;
        int infection_day=community[tid].infection_day_seasonal;
        switch (profile)
        {
            case 1:
                a = d_gamma1[infection_day]; break;
            case 2:
                a = d_lognorm1[infection_day]; break;

68
case 3:
    a = d_weib1[infection_day]; break;
case 4:
    a = d_gamma2[infection_day]; break;
case 5:
    a = d_lognorm2[infection_day]; break;
case 6:
    a = d_weib2[infection_day]; break;
}
if(a!=0)
    p = ((repnum/(((1-asymp)*percent_symptomatic)+asymp))*a)/
        ((community[tid].contact_counter_hh*k_hh)+
         (community[tid].contact_counter_wp*k_wp)+
         (community[tid].contact_counter_er*k_er));
}
if (z==-1)
{
    z=curand_uniform(&st);// uni(0,1);
}
if (z <= (k*p))
{
    if (community[id].prior_virus==0)
        //no prior infection, infected with seasonal
        PRNG=curand_uniform(&st);
        if(PRNG <= percent_symptomatic)
        {
            community[id].symptomatic=1;
            value= curand(&st);
            value%=3;
            if(value==0)
                value=3;
            community[id].profile_seasonal=value ;
        }
    else
    {
        community[id].symptomatic=2;
        value=curand(&st);
        community[id].profile_seasonal=(value %3)+4;
    }
    community[id].disease_clock_seasonal = 0;
    community[id].generation_seasonal = community[tid].generation_seasonal + 1;
    community[tid].rn_seasonal = community[tid].rn_seasonal + 1;
    community[id].infection_day_seasonal = -2;
    community[id].day_begin_infection_seasonal = day;
    community[id].virus = 2;
}
community[id].prior_virus = 2;
community[id].rn_init_seasonal = reproduction_number_seasonal;
community[id].total_infected_seasonal+=1;
community[id].total_infected_seasonal_only+=1;
}
if (community[id].prior_virus==1 && community[id].disease_clock_pandemic>=0)
{//Infected with pandemic, still in the pandemic process and infected with seasonal
PRNG=curand_uniform(&st);
if(PRNG <= percent_symptomatic)
{
    community[id].symptomatic=1;
    value= curand(&st);
    value%=3;
    if(value==0)
    value=3;
    community[id].profile_seasonal=value ;
}
else
{
    community[id].symptomatic=2;
    value=curand(&st);
    community[id].profile_seasonal=(value %3)+4;
}
community[id].disease_clock_seasonal = 0;
community[id].generation_seasonal = community[tid].generation_seasonal + 1;
community[tid].rn_seasonal = community[tid].rn_seasonal + 1;
community[id].infection_day_seasonal = -2;
community[id].day_begin_infection_seasonal = day;
community[id].virus = 3;
community[id].prior_virus = 3;
community[id].rn_init_seasonal = reproduction_number_seasonal*epsilon_s;
community[id].total_infected_seasonal+=1;
community[id].total_coinfected_p_s+=1;
}
if (community[id].prior_virus==1 && community[id].disease_clock_pandemic==3 &&
    community[id].infection_day_pandemic==5)
{//Infected with seasonal, recovered from seasonal process and infected with pandemic
PRNG=curand_uniform(&st);
if(PRNG <= percent_symptomatic)
{
    community[id].symptomatic=1;
    value= curand(&st);
    value%=3;
    if(value==0)
    value=3;
}
community[id].profile_seasonal = value;
}
else
{
    community[id].symptomatic = 2;
    value = curand(&st);
    community[id].profile_seasonal = (value % 3) + 4;
}
community[id].disease_clock_seasonal = 0;
community[id].generation_seasonal = community[tid].generation_seasonal + 1;
community[tid].rn_seasonal = community[tid].rn_seasonal + 1;
community[id].infection_day_seasonal = -2;
community[id].day_begin_infection_seasonal = day;
community[id].virus = 2;
community[id].prior_virus = 3;
community[id].rn_init_seasonal = reproduction_number_seasonal * epsilon_sr;
community[id].total_infected_seasonal += 1;
community[id].total_reinfected_p_s += 1;
}
}
//end of virus 2
}
community[tid].contact_counter = 0;
community[tid].contact_counter_hh = 0;
community[tid].contact_counter_wp = 0;
community[tid].contact_counter_er = 0;
state[(tid % (128 * 128))] = st;
tid += blockDim.x * gridDim.x;
}

__global__ void diseaseSpreadWend(struct entity *community, struct bus *business,
                                   unsigned long start, unsigned long size, int hour,
                                   curandState *state, unsigned char *contact_rate)
{
    unsigned long tid = start + threadIdx.x + blockIdx.x * blockDim.x;
    unsigned long location;
    int contactRate, Rate, count;
    curandState st;
    while (tid < size)
    {
        location = community[tid].schedule[hour];
        count = business[location].count_total;
        st = state[(tid % (128 * 128))];

    }
if(hour==community[tid].errandw[0] || hour == community[tid].errandw[1] ||
    hour==community[tid].errandw[2])
{
    contactRate= contact_rate[location];
    if(business[location].count_total <= contactRate)
    {
        contactRate=business[location].count_total-1;
    }
    if(contactRate > 0)
    {
        for (int i = 0; i < contactRate; i++)
        {
            Rate = curand(&st); //contact_business(i,b);
            Rate = Rate % count;
            while(business[location].array_id[Rate] == community[tid].id)
            {
                Rate = curand(&st);
                Rate = Rate % count;
            }
            community[tid].contact_array_id[community[tid].contact_counter+i]=
                business[location].array_id[Rate];
        }
        community[tid].contact_counter += contactRate;
        community[tid].contact_counter_wp +=contactRate;
    }
}
state[(tid%(128*128))]=st;
tid+= blockDim.x * gridDim.x;
}
References


