GPGPU Parallelization of Self-Calibrating Agent-Based Influenza Outbreak Simulation

Peter Holvenstot
College of Eng. and Applied Sciences
Western Michigan University
Kalamazoo, MI 49008–5466
Email: peter.holvenstot@gmail.com

Professor Diana Prieto
College of Eng. and Applied Sciences
Western Michigan University
Kalamazoo, MI 49008–5466
Email: diana.prieto@wmich.edu

Professor Elise de Doncker
College of Eng. and Applied Sciences
Western Michigan University
Kalamazoo, MI 49008–5466
Email: elise.dedoncker@wmich.edu

***Abstract*—Agent-based simulations of influenza spread are useful for decision making during public health emergencies. During such emergencies, decisions are required in cycles of less than day, and agent-based models should be adapted to support such decisions. The most important considerations for model adaptation are fast calibration of the model, low computational complexity as the population size is scaled up, and dependability of the results with low replication quantity. In previous work, we presented a self-calibrating model for agent-based influenza simulations. We now investigate whether general-purpose GPU computation is effective at accelerating the processing of this model to support health policy decision-making for pandemic and seasonal strains of the virus. The results of this paper indicate that a speedup of 94.3x is obtained with GPU algorithms for simulation sizes of 50 million people. Our GPU implementation scales linearly in the number of people which makes it a good choice for real-time decision support.**

I. Introduction

There is a need for informative simulations to support health policy decision-making during emergent disease outbreaks. Severe outbreaks may require a policy response from public health officials, and models have proven to be useful to project the effects of proposed policies [1], [4].

Several models have been proposed for influenza decision support. Diffusion-based models offer rapid processing times by coarsening the granularity of the simulation. These models assign individuals into compartments, where in each compartment every individual makes the same number of contacts and a contact can be any individual in the compartment [2]. These simulations may be run quickly, but are less informative because they do not model individual behavior. Agent-based simulations model the spread of the disease through interactions between specific individuals in the contact network which allows greater ability to model individual behavior and complex mitigation strategies. Agent-based simulation are computationally expensive as they require a complex calibration process in which its parameters are adjusted to field observation data. Their computational burden grows as the population in the outbreak increases, and due to the stochastic nature of these simulations, many replicates must be run in order to ensure statistical confidence of the results.

As the spread of the disease increases, effective response becomes more difficult and models must provide statistically confident results as quickly as possible. Due to their computation cost, agent-based models are yet to be adapted to support operational decisions that generally occur in cycles of less than 4 to 6 hours [2], [3]. This implies several design considerations: 1) Models should be calibrated as fast as possible. In addition, 2) models need to run extremely fast, even with the increasing size of an outbreak. And 3) models should be replicated quickly.

In previous work, we have addressed consideration 1) by presenting an agent based model which uses observed epidemiological information to calibrate simulation parameters [5]. In this paper, we address considerations 2) and 3) by presenting a GPU accelerated version of our model and investigating its performance under different GPU environments and outbreak sizes. We also discuss whether our simulation is a good candidate for parallelization using GPUs.

Several approaches have been proposed for accelerating general purpose simulations of influenza outbreaks. DiCon [6], ABM++ [7], and FluTE [8] are examples of MPI implementations using CPUs. The software presented in [9] is an example CPU/GPU hybrid approach. Our approach differs because it is implemented entirely in the GPU memory of a single device, which offers a greater speedup. Our paper contributes a modeling framework capable of supporting rapid decisionmaking cycles during influenza emergencies.

In this paper, we investigate whether this model may be effectively accelerated by parallel algorithms using GPUs. Section II describes the operation of the self-calibrating agent-based model. Section III describes the features and limitations of the GPU architecture. Section IV describes our GPU implementation. Section V describes the performance of our model across different simulation configurations and hardware.

II. Description of Simulation

The simulation is an agent-based model of co-circulating influenza strains which incorporates a self-calibrating reproduction number. In the model, agents transition independently between Susceptible, Infected, and Recovered states. At initialization, all agents begin in the susceptible status. A small random initial population of agents is selected to initiate the outbreak. Agents with an infected status make contacts with other agents, which represent possible opportunities for the disease to spread. After a culmination period of 10 days, agents

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

transition to a recovered status. This status data is maintained separately for both pandemic and seasonal strains of the virus.

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

For each strain, a basic reproduction number is given as an input parameter. This represents the average number of further infections that an infected person will generate throughout their period of infectiousness, assuming that the entire population is susceptible. Another parameter given is viral shedding, which is used as a proxy for infectiousness. Carrat et al provide measurements of viral shedding occurring on each day between infection and recovery [10], and we express these daily values as a fraction of the total viral shedding throughout this period. This value is used to distribute the expected number of infections given by the basic reproduction number throughout the period of infectiousness, yielding an expected number of infections for each day [5].

The expected daily reproduction number is further distributed through the daily contacts of an infected case, which have been selected from the simulation. This yields a probability of successful transmission to each contact.

After the simulation has run for a specified number of days, a reproduction number is calculated for each generation of each strain. The reproduction number $\widehat{R}_o$ for generation k is calculated by the ratio between the number of infected cases in the generation $k+1$ and the number of infected cases in a generation k. We conclude that the model is calibrated when the basic reproduction number introduced to the simulation is statistically similar to the maximum value of $\widehat{R}_o$ obtained across all generations. This calibration approach has been further tested and explored in [5].

A few optimizations are used in both the GPU implementation as well as the reference CPU implementation. Only hours in which contacts will be made are generated and simulated. As contacts are strictly unidirectional from the infector to the victim, agents who do not have an active infection have no chance of spreading disease and do not make contacts. Workplaces and households remain constant throughout the simulation, so these locations are generated once and reused throughout the simulation.

III. GPU Architecture

GPUs are specialized processors which process data in a highly parallel fashion. Originally designed to handle the computational loads of graphics rendering, the advent of general-purpose GPU computation languages such as CUDA and OpenCL has allowed them to be applied to many other algorithms. GPUs achieve high computational throughput by using group of cores to perform the same operation on many pieces of data. High-end devices offer thousands of cores, and even low-end consumer devices often offer several hundred cores. This extreme degree of parallelism does come at a cost, and not all algorithms can be effectively accelerated on this architecture.

GPU cores are controlled on several levels of granularity. Cores are assigned to computational groups called blocks, which can share on-chip memory to perform tasks cooperatively. Within a block, threads are controlled in smaller groups called warps that execute instructions and memory accesses in lockstep. Many blocks can be resident on a multiprocessor at once, which allows cores to be used for other tasks in order to hide the latency of global memory access. These blocks are used in a larger group called a grid. Blocks and grids may be 1D, 2D, or three-dimensional and these dimensions may be controlled by the user.

Memory usage is the most important design consideration in the GPU implementation of this algorithm. CPUs are frequently equipped with very large amounts of memory, on the order of 128GB to 512GB or more. However, GPUs have a very limited amount of onboard memory. Most commodity desktop GPUs are equipped with only 1-2GB of memory, and even very expensive GPGPU compute cards only have 5-12GB. Additionally, memory bandwidth is a critical limit in algorithm performance. Thus, making efficient use of memory is paramount in order to scale the simulation to large levels.

A variety of other factors make it difficult to reach full memory utilization. If the GPU is used as a display device, resources allocated to this task cannot also service the program. Memory must be contiguous blocks, and fragmentation can result from external sources or poor memory usage within the program. Error-corrected memory is available on Tesla compute cards, but causes a 10% overhead to store the correction codes. Many parallel implementations of sort algorithms require additional auxiliary global memory as working space.

IV. GPU implementation

We implemented the simulation using the CUDA framework. Several portions of the CPU implementation were changed to better fit GPU architecture. Moving individuals to locations dominates the program runtime in both implementations, so this is a critical portion where good performance must be achieved. In the CPU implementation, locations were implemented as vectors in which personIDs were stored. With a reasonable number of threads, this can be parallelized using mutual-exclusion locks and other techniques. This approach can be generally described as a number of cores working independently to accomplish the task load. However, this approach is unsuitable for the GPU synchronization and memory models.

Instead, the GPU implementation uses groups of threads working together within a flat memory space. The task of location generation is broken into two parts which can both be parallelized effectively. First, a sort algorithm is used to sort personID numbers, with the scheduled location number used as a key. Each location is now represented as a contiguous region of personIDs somewhere within a large array. A vectorized lower-bound search is used to identify the array index of the first person scheduled at a given location number, which is referred to as the location offset. The offset of location l and the offset of location $l+1$ together define the bounds of the region within the array.

Schedules are represented by a location ID and an hour (where appropriate). The encoding of the schedule is found to have a high impact on algorithm performance. In the initial implementation, this was represented as a tuple (hour,locationID), but this performed extremely poorly when sorting. Instead, these are combined into a scheduleID value of (hour * number_locations) + location_id that combines both pieces of data. A scheduleID can be viewed as representing one location at one hour. This approach provides greatly improved performance and can be handled by any sort-by-key algorithm.

Initially some of the setup and the calculation of the final outputs were implemented on the CPU. However, as the runtime of the main algorithm improved this became a significant factor in overall program runtime and necessitated implementation on the GPU.

The final reproduction calculations are accomplished in a manner similar to generating locations. All of the generation numbers are sorted, and vectorized binary searches are used to determine the number of people in each generation. This data is then transferred to the CPU, where the final reproduction values are calculated.

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

A. *Pseudo-Random Number Generator*

As a stochastic simulation, this program consumes a large amount of random numbers in order to generate schedules and select contacts. This requires the use of a pseudorandom number generator (PRNG) within the simulation. We elected to use a counter-based PRNG due to several advantages they offer, and we selected the Random123 library for our implementation [11].

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

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

The counter value can be chosen in ways that provide additional advantages. In our simulation, the output value of a PRNG generation is dependent on the position of the data item being processed. If an algorithm requires C random numbers per data item processed, the first counter input value for each item can be expressed as
ctr = globalOffset + (dataIndex * C)
First, this means that the RNG stream for the program is hardware-independent. Arbitrary numbers of cores can be applied in any compute grid configuration, which allows flexibility in tuning performance across varying hardware.

Additionally, the global offset value used to generate a sequence of RNG numbers can be stored and used in later parts of the algorithm. This can be used to re-generate arbitrary sections of the RNG stream with low computational complexity. This allows us to trade abundant processor power for scarce global memory in certain situations.

B. *Approaches*

The implementation went through three distinct phases with increasing scalability.

In the initial implementation, global memory was used to store all data. Workplace, household, and errand contacts were generated separately, and the contacts for each hour were output sequentially into an array. At the end of the day, these were sorted by infector, and processed into actions. A series of operations filtered contacts whose target was not susceptible, sorted them by target, and then removed duplicate actions. The actions were then performed on the status arrays. However, storing all the contacts used a very large amount of global memory and the simulation could not be scaled very large.

Instead, to maximize performance and scalability our second phase focused on avoiding the use of global memory for the contacts array. Contacts are generated for each individual for an entire day at once, and staged in shared memory during the contact-making and processing phases. When a successful infection attempt is made, the status of the target individual in global memory is immediately updated using atomic compare-and-swap operations. This can cause a data hazard if the target is already infected with one strain of the virus and has not yet been processed, as the day of their infection may be invalid. Thus, some additional checking is required when processing contacts. This approach offers much better scalability and speedup than the previous.

The final phase focused on eliminating data stored about infected individuals. The contact algorithm needs to know at which locations an individual resides in order to select contacts. This data would no longer be easily accessible after sorting, so previous approaches copied this data out to a lookup table for infected individuals.

While the algorithm must have this scheduling data, an alternate approach is to reconstruct it instead of saving it. Since the output of the PRNG is associated with the position of a data item rather than a PRNG state, if the contact algorithm knows the value of the global PRNG offset when the schedule was generated then it can re-generate the PRNG output used to assign a location for a particular individual. This results in trading processor cycles for global memory.

```
read_inputs()
for i=0 to num_households do
    assignHouseholdPopulation(i)
end for
allocate_arrays()
for i=0 to num_people do
    assignAgeAndWorkplace(i)
end for
setupFixedLocations()
```

Fig. 1. Setup Algorithm

This re-generation approach cannot be used on household locations because of the manner in which the simulation is initialized. Simulation size is specified in terms of a number of households, and the number of people in the simulation is determined by assigning household populations according to a PDF. However, another trick can be used when selecting contacts from households. Since household numbers are assigned sequentially and then personIDs are assigned based on this, the ith index of the household people table is always equal to i. This allows us an equal savings of memory in a different area.

Taken together, these approaches yield an approximately 25% reduction in memory usage while having a nearly negligible effect on simulation time.

C. Pseudocode

Our algorithm consists of a setup phase, a daily loop that runs the simulation, and a final calculation phase. The setup phase shown in Figure 1 loads the parameters of the experiment and generates the population. Population size is controlled by specifying the number of households that will be generated, which is given as an input parameter. A PDF is used to assign each household a type, which specifies the number of adults and children in that household. Once the population size of the simulation is known, the program allocates the global memory arrays, and then assigns data to each individual. Adults are assigned to a workplace according to a PDF. Children are assigned a specific age, and this age is used to randomly select an age-appropriate school as their workplace. Since workplaces and households remain constant throughout the simulation, these locations are treated as a special case for scheduling. They are generated once and re-used throughout the simulation.

The daily loop algorithm shown in Figure 2 controls the state of the simulation. First, all population members are iterated to transition agents who have reached culmination from infected to recovered status. This same loop is used to count the disease state of each agent for each strain, which is used as an output. Schedule items are generated for each agent, and then these are used as keys to sort the personIDs by location. Finally, a selection algorithm is used to select all individuals with an active infection. The contacts kernel then handles the process of selecting contacts and transmitting infection.

The contact selection and contact processing algorithms take place within a kernel and contact data is stored in shared memory. In the selection phase, each individual randomly selects contacts from their scheduled locations/hours

```
countAndUpdateStatus()
assignSchedules()
generateLocations()
activeInfections[]← selectInfected()
for all myIdx in activeInfections do
    selectContacts()
    processContacts()
end for
```

Fig. 2. Daily Loop Algorithm

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

Fig. 3. Contact Selection Algorithm

according to an arbitrary algorithm. Individuals may not select themselves as contacts and if this occurs the next sequential person at the location (with wraparound) is chosen instead. Each contact is assigned an arbitrary kappa value representing the relative opportunity for disease to spread. The algorithm outputs an array of personID values selected as contacts and an array of kappa values for the contact.

Figure 4 gives a simplified pseudocode for the processing phase. Our implementation calculates separate infection thresholds, yVals, and transmission success for both pandemic and seasonal strains of the virus, but we have abbreviated this due to space limitations. In the self-calibration step, the kappa sum, base reproduction numbers, viral shedding profiles, and individual epistemological data are used to calculate the probability of a successful infection for a contact with a kappa value of 1. This base probability is then modified by the kappa value of each contact.

V. Results

The GPU implementation was evaluated on several different devices reflecting a variety of GPGPU-capable hardware. The NVIDIA K20 is a high-end Tesla compute card targeted at

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

Fig. 4. Contact Processing Algorithm

high-performance computing. The Nvidia GT640 is a modern but low-end commodity desktop graphics card intended for media PC usage. Finally the Quadro FX 880M is a relatively old chipset used in workstation laptops. The program was built with full compiler optimizations in each configuration. The most important specifications when comparing the performance of these devices is are the number of cores and the memory bandwidth, which are given in Table I. The final column of the table gives an approximate maximum population scale value which can be supported on the device, as described below.

To provide a comparison, we measured the performance of a CPU implementation on a server using dual 8-core 2.6GHz Xeon E5-2670 CPUs in Western Michigan University's High Performance Computational Science Laboratory. We provide results for a single-threaded implementation as well as a multi-threaded implementation using OpenMP. These implementations use the Mersenne Twister PRNG from the Boost.Random library. In the OpenMP version, each thread uses its own independent PRNG instance.

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

Table III shows the speedup of the GPU and OpenMP implementations relative to the single-threaded implementation. GPU speedup can be given as kernel speedup, which describes the increase in performance of accelerated portions of the program, and program speedup, which describes the improvement in overall program time from accelerating certain portions. As we have accelerated the entire simulation onboard the GPU, our speedups for the GPU simulation reflect program speedup. Profiler measures were not included as they were not informative of the overall simulation performance. Sorting the schedule items is the most intensive step of the simulation and consumed a majority of the runtime.

The number of people in the simulation is the dominant factor in the runtime. The CPU can complete small simulations reasonably quickly. However, as the simulation is enlarged its performance begins to degrade slightly. Within its limits the GPU provides excellent performance and a speedup is observed at all simulation sizes on all devices. The speedup of the GPU implementation increases as the simulation is scaled up until the memory limits of the device are reached. This behavior is likely due to fixed launch/synchronization overhead, which is amortized across larger numbers of elements as the simulation scale increases.

TABLE I. DEVICE SPECIFICATIONS

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

TABLE II. RUNTIME COMPARISON

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

TABLE III. SPEEDUP

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

VI. FUTURE WORK

A. Simulation Model

While this implementation is focused on accelerating an existing simulation model, our work provides a generalized framework for accelerating self-calibrating agent-based contact models using a large number of individuals and locations. This could be used to accelerate other simulation models with the use of a more complex scheduling algorithm.

The current model considers all locations within a location type to be equally probable destinations. This is a reasonable assumption when simulation size is small, but within a larger area individuals would be more likely to visit certain locations than others. One possible model could assign greater probability to certain locations of a given type. This would cause more individuals to be scheduled into those locations, reflecting a difference in population density between urban and rural areas.

A more complex model could assign additional data to each individual which could be used in the scheduling process. For example, a parameter representing the person's geographic location could be used to provide greater weight to locations near to them. Similar data about individuals is already used by the scheduling algorithm, for example to determine whether to schedule an errand (adults) or an afterschool activity (children).

B. Simulation Performance

The runtime of the simulation might be further decreased by using other sorting algorithms. Since the sort operation used when generating locations consumes a large percentage of the program runtime as well as a great deal of memory, improvements to this part of the program have a large impact on simulation runtime and size limits. GPU parallel sorting is an area of active research and incorporating recent findings will have direct performance benefits in a critical portion of the algorithm. One possible approach would be to pre-process the scheduling data in shared memory immediately after generation. Since the re-generation approach allows the schedule of an individual to be reconstructed, the locations of infected individuals do not need to be explicitly stored. Rather than writing the errand sequence directly into global memory ordered according to the personID, the errands can be written into shared memory and the block can pre-sort the errand sequence into a tile. The tile can then be written into global memory, where a mergesort could be used to sort the tiles. This removes a complete round-trip to global memory, which may offer performance advantages over a naive approach.

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

C. Simulation Scale

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

It would also be desirable to increase the size of the simulation much further than incremental gains will allow. While the maximum size of the simulation is limited by the available GPU memory, multiple GPUs could be used in a single machine or within a cluster. Each device is capable of simulating a large population center, and with the addition of an algorithm to control travel between these units national- or international-scale simulations could be rapidly completed. Low-end devices have an excellent cost-to-memory ratio and the performance remains high, so these devices may be a cost-effective approach to creating large simulations.

Another approach would be to shift the parts of the program that are highly memory-intensive back to a CPU implementation. The GPU could be utilized as a dedicated coprocessor for generating schedules and sorting the location arrays. These tasks are compute-intensive and could be accomplished efficiently by the GPU using a relatively small amount of data, then transferred to the CPU for use in making contacts. Very large simulations could be supported by splitting these tasks into segments capable of fitting into GPU memory and then merging the segments within host memory. Since the GPU parallelization model is generally more restrictive than CPUs, the CPU portions could easily be parallelized using an arbitrary number of cores within a shared memory space.

VII. Conclusion

Our parallel GPU implementation provides substantially greater speedups than a multi-threaded CPU implementation, and can be scaled to support large simulations when executed on both high-end GPGPU devices and commodity hardware. Our agent-based influenza model is a good candidate for GPU acceleration due to its high degree of parallelism. This allows informative simulations to be completed rapidly to support health policy decision-making during emergent outbreaks.

Acknowledgment

The authors would like to thank the Western Michigan University High-Performance Computational Science Laboratory, established with support under NSF grant 1126438 and the CUDA Teaching Center Grant from NVIDIA.

References

[1] Interim pre-pandemic planning guidance: community strategy for pandemic influenza mitigation in the united states. http://www.pandemicflu.gov/plan/community/communitymitigation.pdf

[2] D. Prieto and T. Das and A. Savachkin and A. Uribe and R. Izurieta and S. Malavade, *A systematic review to identify areas of enhancements of pandemic simulation models for operational use at provincial and local levels.*, BMC Public Health, 2012, 12:251.

[3] *Study to determine the requirements for an operational epidemiological modeling process in support of decision making during disaster medical and public health response operations*, Tech. rep., Yale New Haven Center for Emergency Preparedness and Disaster Response and US the Northern Command (2012)

[4] M. Tizzoni and P. Bajardi and C. Poletto and J. Ramasco and D. Balcan and B. Gonalves and N. Perra and V. Colizza and A. Vespignani, *Real-time numerical forecast of global epidemic spreading: case study of 2009 A/H1N1pdm*, BMC Medicine, 2012, 10:165.

[5] D. Prieto and T. Das, *An operational epidemiological model for calibrating agent-based simulations of pandemic influenza outbreaks*, Health Care Manag Sci. 2014 Apr 8. [Epub ahead of print].

[6] *Dicon: Disease Control System* [http://www.bio.utexas.edu/research/meyers/dicon/].

[7] MIDAS, *ABM++* [https://mission.midas.psc.edu/].

[8] D. Chao and M. Halloran and V. Obenchain and I. Longini, *FluTE, a publicly available stochastic influenza epidemic simulation model*, PLoS Comput Biol 2010, 6:1-8.

[9] K. Bisset and A. Aji and M. Marathe and W. Feng, *High-performance biocomputing for simulating the spread of contagion over large contact networks*, BMC Genomics 2012, 13 (2):S3.

[10] F. Carrat and E. Vergu and N. Ferguson and M. Lemaitre and S. Cauchemez and S. Leach and A. J. Valleron , *Time Lines of Infection and Disease in Human Influenza: A Review of Volunteer Challenge Studies*, Am. Jnl of Epidemiology 2008, 7:775-785.

[11] Salmon, J. K. and Moraes, M. A., *Random123: a Library of Counter-Based Random Number Generators*, [http://deshawresearch.com/resources_random123.html].