High-Performance Biocomputing for Simulating the Spread of Contagion over Large Contact Networks

Keith R. Bisset
Virginia Bioinformatics Inst.
Virginia Tech
Blacksburg, Virginia, USA
kbisset@vbi.vt.edu

Ashwin M. Aji
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
aaji@cs.vt.edu

Madhav V. Marathe
Virginia Bioinformatics Inst.
Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
mmarathe@vbi.vt.edu

Wu-chun Feng
Dept. of Computer Science
Dept. of Elec. & Comp. Eng.
Virginia Tech
Blacksburg, Virginia, USA
feng@cs.vt.edu

***Abstract*—Many important biological problems can be modeled as contagion diffusion processes over interaction networks. This paper shows how the EpiSimdemics interaction-based simulation system can be applied to the general contagion diffusion problem. Two specific problems, computational epidemiology and human immune system modeling, are given as examples. We then show how the graphics processing unit (GPU) within each compute node of a cluster can effectively be used to speed-up the execution of these types of problems by an additional 3.3-fold over a CPU-only solution.**

***Keywords*-Computational Epidemiology; Human Immune System Modeling, Graphics Processing Unit; CUDA.**

I. Introduction

Network-based models are natural generalizations of stochastic mass action models to stochastic dynamics on arbitrary interaction networks. Consider, for example, a typical network model for computational epidemiology:

- a network with labeled vertices and edges, representing people and possible disease transmission paths, respectively;
- vertex labels, representing the corresponding person's state of health, i.e., susceptible (S), infectious (I), or recovered (R);
- edge labels, representing the probability of transmission from one person to another; and
- discrete-time dynamics, corresponding to percolation across the graph, i.e., the label on a vertex in state S changes to I based on independent Bernoulli trials for each neighbor in the state I with the probability specified by the edge between them; the label on a vertex in state I changes to R with some fixed probability.

Our interaction-based network will model diffusion processes for large numbers of agents (10^6 – 10^9) in a variety of domains [1]. It was originally created to model the spread of infectious diseases through large human populations [2]. It has since been adapted to also model the spread of malware in wireless networks [3], and the spread of information, fads, and norms through friendship networks. Work is currently underway to model the human immune responses to two gastroenteric pathogens that will require simulating up to 10^9 individual cells [4]. We have successfully computed the spread of influenza on a network representing the population of the United States (270 million people and 1.4 billion edges) on a CPU cluster with 768 cores.

Today, computational performance improvements are increasingly achieved through parallelism within a chip, both in traditional multi-core architectures as well as the many-core architectures of the graphics processing unit (GPU). Amongst the most prominent many-core architectures are the GPUs from NVIDIA and AMD/ATI, which can accelerate a large variety of scientific applications, at very affordable prices [5]–[7]. Our previous work focused on reducing the communication overhead and improving the inter-node performance of our simulation systems [8]. The work described in this paper focuses on improving the intra-node performance through the use of GPU accelerators.

Our agent-based models are built on rigorous mathematical foundations – one of the unique aspects of our work. These are called Graphical Discrete Dynamical Systems [9]. The mathematical model consists of two parts: (i) a co-evolving graphical discrete dynamical system framework (CGDDS) that captures the co-evolution of system dynamics, interaction network and individual object behavior, and (ii) a partially observable Markov decision process that captures various control and optimization problems formulated on the phase space of this dynamical system. Informally speaking, a CGDDS consists of the following components: a dynamic graph $G_t(V_t, E_t)$ in which vertices represents individual objects (agents) and edges represent a causal relationship that is usually local, a set of local state machines (automata), one for each vertex specifying the behavior of the agents and a set of update functions, one per edge that describes how an edge can be modified as a function of the state of its endpoints. The state and behavioral changes of individual objects are a function of their internal state and its interaction with neighboring agents. These neighbors change over time and space, and thus the model needs to explicitly

978-1-61284-852-5/11/$26.00 ©2011 IEEE

represent this network evolution.

CGDDS serve as a bridge between mathematical simulation theory and HPC design and implementation. Like state charts and other formal models, they can be used for formal specification, design and analysis of multi-scale biological and social systems. CGDDS extend the algebraic theory of dynamical systems in two important ways. First, we pass from extremely general structural and analytical properties of composed local maps to issues of provable implementation of sequential dynamical systems in computing architectures, and to specification of interacting local symbolic procedures generally. This is related to successive reductions of CGDDS to procedural primitives, which leads to a notion of CGDDS-based distributed simulation compilers with provable simulated dynamics (e.g., for massively parallel or grid computation). Second, the aggregate behavior of iterated compositions of local maps that comprise a CGDDS can be understood as a (specific) simulated algorithm together with its associated and inherent computational complexity. We have called this the algorithmic semantics of a CGDDS (equivalently, the algorithmic semantics of a dynamical system or a simulation). It is particularly important to view a composed dynamical system as computing a specifiable algorithm with provable time and space performance.

A. Computational Epidemiology

The threat of a global disease outbreak, such as pandemic influenza, is an important public health problem facing the world. The current potential for a H1N1 or H5N1 pandemic underscores the conspicuous risk to public health and global economy. To plan and respond proportionately to such pandemics, public health officials need a systematic assessment of the socio-economic and health impact of the disease, interventions, and other mitigation efforts [10], [11]. Policy makers desire an understanding of intervention possibilities and pitfalls for limiting pandemic risk and assisting vulnerable populations. These interventions may include social distancing, school and workplace closures, and the use of pharmaceuticals.

Following Beckman et al. [12], we estimate a contact network in four steps: (1) population synthesis, in which a synthetic representation of each household in a region is created; (2) activity assignment, in which each synthetic person in a household is assigned a set of activities to perform during the day, along with the times when the activities begin and end; (3) location choice, in which an appropriate real location is chosen for each activity for every synthetic person; and (4) contact estimation, in which each synthetic person is deemed to have made contact with a subset of other synthetic people.

The contact network is a graph whose vertices are synthetic people, labeled by their demographics, and whose edges represent contacts determined in step four, labeled by conditional probability of disease transmission.

Each of the four steps makes use of a stochastic model; the first three incorporate observed data for a particular region. The synthetic population is generated using an iterative proportional fit to joint distributions of demographics taken from a census [13]. It consists of a list of demographic variables such as age, gender, income, household size, and education. Activities like school, work, and shopping are assigned using a decision tree model based on household demographics fitted to activity or time-use survey data. This step creates a "personal day planner" for each person in the synthetic population. Activity locations are chosen based on a gravity model and land use data. That is, every synthetic activity produced in step three is assigned to an actual location – office building, school, shopping mall, etc. – based on its distance from the person's previous activity and its attractiveness, a measure of how likely that the activity happens there.

Population synthesis uses a non-parametric, data-driven model based on US census data for the year 2000, including the Public Use Microdata Sample [14]. The actual activity templates assigned to households are data-driven, based on the National Household Transportation Survey [15]. The gravity model used for location choice contains nine parameters. The locations' addresses and attractor values are derived from Dun & Bradstreet's Strategic Database Marketing Records.

Disease propagation is modeled by

$$p_{i \to j} = 1 - (1 - r_i s_j \rho)^{\tau} \tag{1}$$

where $p_{i \to j}$ is the probability infectious individual i infecting susceptible individual j, τ is the duration of exposure, r_i is the infectivity of i, s_j is the susceptibility of j, and ρ is the transmissibility, a disease specific property defined as the probability of a single completely susceptible person being infected by a single completely infectious person during one minute of exposure [1].

The estimated contact networks are certainly not correct in the sense that each synthetic person represents an actual person, and that edges in the estimated network have a one-to-one correspondence with edges in the real network. If it were the case that only such a one-to-one match reproduced important dynamical behaviors, it would be futile to estimate the networks, or indeed to build any mathematical models of epidemiology for specific large populations. Instead, it is an estimate of one realization of a typical network, the better the network estimate, the more realistic the results – up to a point determined by the inherent variability of epidemic processes [16].

B. Human Immune System Modeling

Inflammatory bowel disease (IBD) is an immunoinflammatory illness of the gut initiated by an immune response to bacteria in the microflora. The resulting immunopathogenesis leads to lesions in epithelial lining of the colon

through which bacteria may infiltrate the tissue causing recurring bouts of diarrhea, rectal bleeding, and malnutrition. In healthy individuals such immunopathogenesis is avoided by the presence of regulatory cells that inhibit the inflammatory pathway. Highly relevant to the search for treatment strategies is the identification of components of the inflammatory pathway that allow regulatory mechanisms to be overridden and immunopathogenesis to proceed. In vitro techniques have identified cellular interactions involved in inflammation-regulation crosstalk. However, tracing immunological mechanisms discovered at the cellular level confidently back to an in vivo context of multiple, simultaneous interactions has met limited success.

We have applied CGDDS formalism to the development of a preliminary model of gut immunity. In this model, individual immune cells make contact and interact with dynamic populations of bacteria and cytokines as they migrate within and among three tissue sites: i) the lumen/lamina propria border, where epithelial cells reside, ii) the lamina propria, more generally termed the effector site of the immune response, and iii) mesenteric lymph node, the inductive site of the immune response. Future modeling efforts will also model the small intestine (and the Peyers's patches) and the gastric mucosa.

This level of detail requires unprecedented scaling on high-performance computing systems – 10^7 to 10^9 cells (agents), simulations with time resolution of minutes for a total period of years and spatial resolution of 10^{-4} meters.

The goals of this model is to build, refine and validate predictive multiscale models of the gut mucosal immune system that compute condition-based interactions between cells for understanding the mechanisms of action underlying immunity to enteric pathogens. The field of translational medicine seeks to estimate effects that behavior observed at the molecular, individual-cell level (in vitro) would have at the multi-cell, tissue level (in vivo/in situ). With this model, one can i) test plausibility of in vitro observed behavior as explanations for observations in vivo/in situ, ii) conduct low-cost, preliminary experiments of proposed vaccines and immunotherapeutics, iii) propose behaviors not yet tested in vitro that could be possible explanations for observations at tissue level, and iv) identify useful areas of research – missing data needed to address a specific hypothesis.

C. EpiSimdemics

EpiSimdemics [2], [8] is an interaction-based, high-performance computing-oriented simulation for studying CGDDS. EpiSimdemics allows us to study the joint evolution of contagion dynamics, interactor (i.e., agent) behavior and interaction networks as a contagion spreads. The interaction network is represented as a bi-partite graph, with interactors (e.g., people or cells) and locations (e.g., building or tissue types) represented as nodes and visits of locations by interactors represented as edges. Each interactor's state is represented as one or more Probabilistic Timed Transition Systems (PTTS). A PTTS is a finite state machine where transitions can be timed and probabilistic.

The computation structure of this implementation, shown in Figure 1, consists of three main components: interactors, locations and message brokers. Given a parallel system with N cores, or Processing Elements (PEs), interactors and locations are first partitioned in a round-robin fashion into N groups denoted by $P_1, P_2, \ldots, P_N$ and $L_1, L_2, \ldots, L_N$, respectively. Each PE then executes all the remaining steps of the EpiSimdemics algorithm on its local data set (P_i, L_i). Each PE also creates a copy of the message broker, MB_i. Next, a set of *visit messages* are computed for each interactor P_i for the current iteration, which are then sent to each location (which may be on a different PE) via the local message broker. Each location, upon receiving the visit messages, computes the probability of spread of contagion for each interactor at that location. Outcomes of these computations, called interaction results, are then sent back to the "home" PEs of each interactor via the local message broker. Finally, the interaction result messages for each interactor on a PE are processed and the resulting state of each affected interactor is updated. All of the PEs in the system are synchronized after each iteration, so that the necessary data is guaranteed to have been received by the respective PEs. The above steps are executed for the required number of iterations (e.g., days in an epidemiology simulation), and the resulting interaction network dynamics are analyzed in detail. The computation that takes place on line 12 of the algorithm consumes roughly 60% of the execution time of the system. It is this portion of the computation that is offloaded to the GPU. Note that this section of the computation is purely serial in that it does not require any inter-process communication. Therefore, any speedup that is gained through this offload will be applied linearly across the entire distributed computation.

II. The NVIDIA GPU Architecture

The NVIDIA GPU (or *device*) consists of a set of streaming multiprocessors (SMs), where each SM consists of a group of scalar processor (SP) cores with a multi-threaded instruction unit. The actual number of SMs vary depending on the different GPU models. The NVIDIA Tesla S2050, which we used for our experiments, consists of 14 SMs, each with 32 SP cores, making a total of 448 SP cores.

All the SMs on the GPU can simultaneously access the *device memory*, which can go up to 4 GB in capacity. The device memory can also be read or written to by the *host* CPU processor. Each SM has faster on-chip *shared memory*, which can be accessed by all the SPs within the SM and are up to 48 KB. The shared memory can be considered to be a software-managed data cache.

The CUDA Programming Model: CUDA (Compute Unified Device Architecture) [17] is the parallel programming

```
1: initialize();
2: partition();                                   ▷ partition data across PEs
3: for t = 0 to T increasing by △t  do
4:     foreach interactor p_j ∈ P_i do            ▷ send visits to location PEs
5:         computeVisits(j, t to t + Δt);
6:         sendVisits(MB_i);
7:     end for
8:     Visits ← MB_i.retrieveMessages();
9:     synchronize();
10:    foreach location l_k ∈ L_i do              ▷ compose a serial DES
11:        makeEvents(k, Visits);                 ▷ turn visit data into events
12:        computeInteractions(k);                ▷ Process Events
13:        sendInteractionResults(MB_i);
14:    end for
15:    MB_i.retrieveMessages();
16:    synchronize();
17:    foreach j ∈ P_i do                         ▷ combine multiple interaction results
18:        updateState(j);
19:    end for
20: end for
```

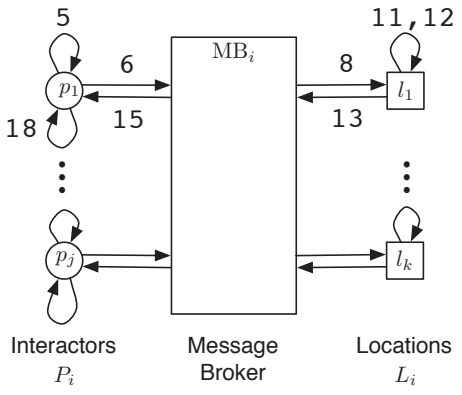

Figure 1. Parallel algorithm (left) and the computational structure of EpiSimdemics. The numbers in the diagram correspond to line numbers in the algorithm.

model and software environment provided by NVIDIA to run applications on their GPUs, programmed via simple extensions to the C programming language. CUDA follows a code offloading model, i.e., compute-intensive portions of applications running on the host CPU processor are typically offloaded onto the GPU device for acceleration. The *kernel* is the portion of the program that is compiled to the instruction set of the GPU device and then offloaded to the device before execution. For this paper, we have used CUDA v3.1 as our programming interface to the GPU.

Each kernel executes as a group of threads, which are logically organized in the hierarchical form of grids and blocks of threads. The dimensions of the blocks and the grid can be specified as parameters before each kernel launch. The kernel is mapped onto the device such that each thread block is executed on only one SM, and a single SM can execute multiple such thread blocks simultaneously, depending on the memory requirements of each block. The on-chip shared memory of an SM can be accessed only by the thread block that is running on the SM, while the global device memory is shared across all the thread blocks in the kernel.

III. EpiSimdemics on a Graphics Processing Unit

As discussed in sections I-C and II, the most computationally intensive part of the application, i.e. *Compute–Interactions* should be *offloaded* to the GPU for faster execution. Since each compute node in our test cluster has a GPU attached to it, every node accelerates the *Compute–Interactions* phase of the algorithm on the dedicated GPU device, while performing the rest of the computations on the CPU. Moreover, since each node in the cluster has identical computation and communication patterns, we can investigate the intra-node parallelization methods for a single GPU device in isolation. More specifically, the GPU offload process can be broken down into the following steps:

1) Transfer the input data from the CPU's main host memory to the GPU's device memory.
2) Invoke the GPU kernel method (*computeInteractions()*), which does the parallel computation on the GPU device.
3) Transfer the output data from the GPU's device memory back to the CPU's main host memory.

We invoke the CUDA kernel once per simulation iteration. Inside each kernel, we loop over all the set of locations that are being processed by the current node, and calculate the contagion spread information for each location for a simulation iteration. The infections are computed for a number of constant time periods among the current occupants of the location, where the contagion transmission is modeled, for example, by Equation 1. For GPU-EpiSimdemics, the locations, the interactors, and their respective PTTS states are transferred to the GPU's device memory as input. After processing, the interaction result messages are transferred back to the CPU's main memory, which are then communicated to the other nodes in the cluster, as necessary. Next, we discuss the mapping of the *Compute–Interactions* phase to the computational and memory units within a GPU.

Mapping to the Computational Units: The threads in the GPU kernel are grouped as blocks in a grid, and parallelism can be exploited at two layers, i.e., independent tasks can be assigned to different *blocks*, while the *threads* in each block can work together to solve the assigned task. We can see that the locations can be processed in parallel, and so the *Compute–Interactions* phase of EpiSimdemics can map very well to the hierarchical CUDA kernel architecture, as shown in figure 2. More specifically, the set of independent

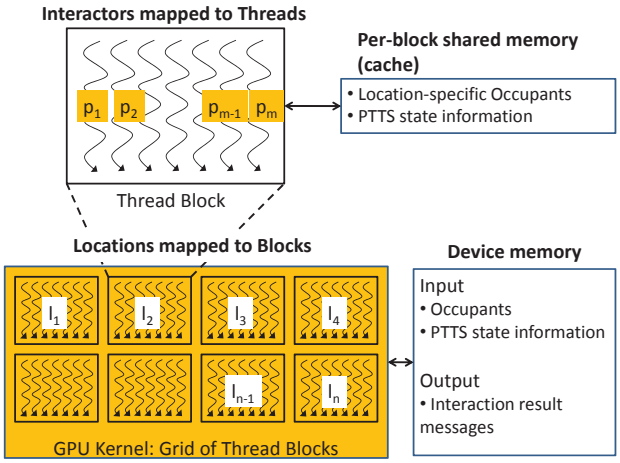

Figure 2. Mapping of *Compute–Interactions* to the NVIDIA GPUs. l_1–l_n denote the set of independent locations, and p_1–p_m indicate the set of interactors within one of the locations.

locations can be assigned to different blocks for parallel processing. The interactors within a location are then assigned to independent threads within the block. The threads (occupants) can cooperatively work together and generate the set of interaction result messages for their location.

Mapping to the Memory Units: All the blocks in a CUDA kernel can access the entire device memory of the GPU. The input data is therefore transferred to the device memory from the CPU. The input data contains information about the occupants in all the locations, and the PTTS state information of each occupant in every location. However, access to the device memory is very slow (400-600 clock cycles [17]) when compared to the faster shared memory or cache memory. Moreover, the faster shared memory is local to each CUDA block, where each block processes a single location. We make use of the faster shared memory by first explicitly copying the relevant occupants and PTTS state information from the device memory to the caches, and then do the processing. The movement of data from the device's global memory to the faster shared memory is slow, but it needs to be done just once per location. The interaction result messages (output data) are directly stored in the device's global memory. Figure 2 shows the mapping of the input data to the different memory hierarchies in the GPU.

We further optimized the device memory accesses and data access patterns in the shared memory, as suggested in the CUDA programming guide [17].

IV. Experimental Analysis

A. Experimental Setup

Data: To validate the performance and scalability of the GPU kernel, we chose a data set that fits in both the CPU and GPU memories within a single node. Our data set was

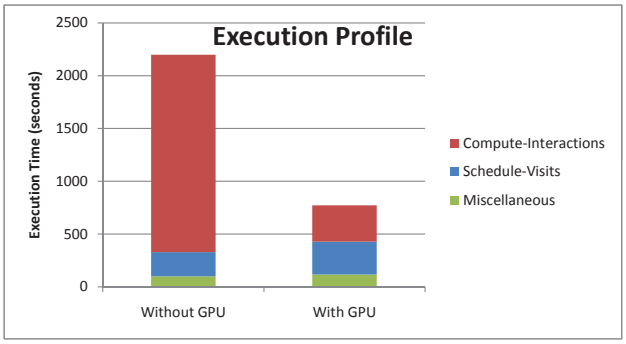

Figure 3. Execution time of different steps in the EpiSimdemics algorithm.

synthetic population from the state of Delaware, consisting of 247,876 persons and 204,995 locations. All our results are based on running the EpiSimdemics for 55 simulated days.

Hardware: We ran our experiments on an eight-node cluster, where each node had a multi-core AMD CPU. Each node was attached to two NVIDIA Tesla S2050 GPUs, which belong to the latest Fermi GPU architecture family. For all our experiments, we used one CPU core and one GPU from each node. Section IV-C describes our ongoing work, where we intelligently use all the available CPU cores and GPU devices within a node to further improve the scalability of EpiSimdemics.

B. Results

Execution Profile: Figure 3 shows the distribution of the different steps of the EpiSimdemics algorithm, for both the CPU and GPU implementations. This result, in particular, was obtained by running the program on a single CPU and accelerating the *Compute–Interactions* part on one GPU. However, the distribution in the execution profile remains nearly the same when run on more nodes in the cluster. Our results show that the intra-node parallelization by using a GPU has been used to accelerate the *Compute–Interactions* phase of EpiSimdemics by up to a factor of *six* over the serial CPU version. The GPU version of the overall application can be up to 3.3-times faster than the CPU-only solution.

Scalability Analysis: In this section, we will show the effect of executing EpiSimdemics on up to eight nodes in the cluster. Ideally, the performance improvement of a program should be proportional to the number of nodes in the system. In practice, we do not get ideal scaling when we increase the number of nodes, because of the overhead incurred by communicating data among the different nodes. However, by accelerating the computation within the node by using a GPU, we can compensate for the performance loss due to communication and improve the scalability of the overall system. Figure 4 shows the performance improvement of EpiSimdemics (with and without the GPU) relative to a single node system. We can see that intra-node parallelization

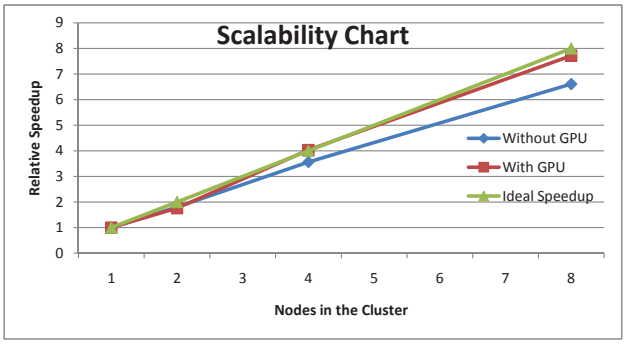

Figure 4. Relative performance of EpiSimdemics with and without the GPU with respect to the ideal speedup.

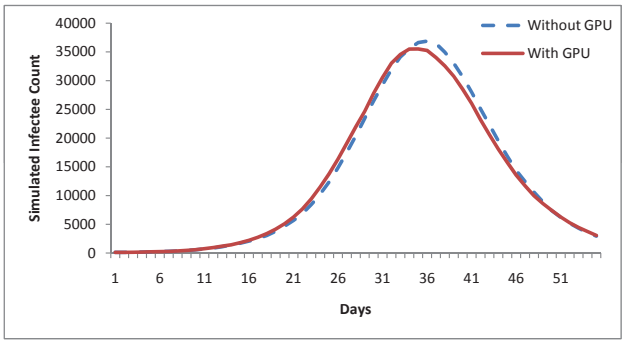

Figure 5. Epi-Curves validating the correctness of GPU-EpiSimdemics.

using GPUs helps in achieving a performance that is very close to the ideal, and is more scalable than the non-GPU solution. Our future work is to optimize the communication overhead to further improve the overall scalability of the program.

Validation of Correctness: We verify the correctness of the contagion spread calculations when we compute that step on the GPU. We do this by plotting the Epi-curves (i.e., new infections per day) that are generated by EpiSimdemics for both the CPU and GPU platforms as shown in Figure 5. We can see that the daily infection distribution is similar on both the platforms, and within inherent variability of the stochastic process being modeled, thereby validating the correctness of GPU-EpiSimdemics.

Insight into Disease Spread: Figure 6 shows a snapshot of the spread of disease in Delaware's Kent and Sussex counties on day 34 of one simulation. The area of interest is divided into small blocks, and each block is colored according to the number of infected individuals who live in that block. Yellow blocks indicate areas of low disease prevalence, while red blocks indicate higher prevalence.

C. CPU-GPU Co-scheduling

The GPU kernel offload method uses the available intra-node parallelism in a GPU and improves the overall scalability of EpiSimdemics. However, this requires each CPU to have a dedicated GPU device to which the tasks can be

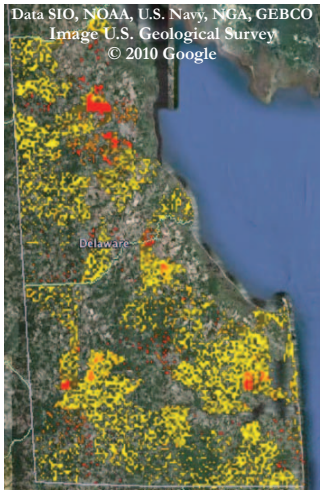

Figure 6. Simulated disease spread on day 34 in Kent and Sussex counties in Delaware. Yellow through red areas indicate increasing disease prevalence.

offloaded for acceleration. But, each node in any present day cluster will usually have more CPUs (2 – 32) than the number of GPUs (1 – 4). For example, our test cluster is made up of 8 nodes with 32 CPU cores per node. Each node is attached to a dual-GPU system, which means that 32 CPU cores in every node are contending with each other to offload the kernels to 2 GPUs. The scalability of EpiSimdemics will be poor if the kernels are offloaded in the default way (by blocking), because CPU resources will be wasted in just waiting for a free GPU. We have explored different kernel offload techniques to improve the scalability of general multi-CPU applications in our ongoing work [18], whose results will be included in the complete version of this paper.

V. Conclusions and Future Work

Interaction-based simulation systems can be used to model disparate and highly relevant problems in biology. We have shown that offloading some of the work in distributed interaction-based simulations can be an effective way to achieve increased intra-node efficiency. We have shown that GPU computing, when combined with effective techniques for inter-node communication, high scalability can be achieved. In the future, we plan to explore ways to exploit multiple cores and multiple GPUs per node by intelligently scheduling work on the two types of processors.

Acknowledgments

We thank our external collaborators and members of the Network Dynamics and Simulation Science Laboratory for their suggestions and comments. This work has been partially supported by NSF Nets CNS-0626964 and CNS-0831633, NSF HSD Grant SES-0729441, NIH MIDAS 2U01GM070694-7, NSF PetaApps Grant OCI-0904844, DTRA R&D HDTRA1-0901-0017, DTRA CNIMS HDTRA1-07-C-0113, DHS 4112-31805, DOE DE-SC0003957, NSF REU Supplement CNS-0845700, US Naval Surface Warfare Center N00178-09-D-3017 DEL ORDER 13,

NSF Netse CNS-1011769, NSF SDCI OCI-1032677 and NIAID & NIH HHSN272201000056C.

REFERENCES

[1] C. Barrett, K. Bisset, S. Eubank, M. Marathe, V. S. A. Kumar, and H. Mortveit, "Modeling and simulation of large biological, information and socio-technical systems: An interaction-based approach," in *Short Course on Modeling and Simulation of Biological Networks*, R. Laubenbacher, Ed. AMS, Jan. 2007, pp. 101–147.

[2] K. Bisset, X. Feng, M. Marathe, and S. Yardi, "Modeling interaction between individuals, social networks and public policy to support public health epidemiology," dec. 2009, pp. 2020 –2031.

[3] K. Channakeshava, D. Chafekar, K. R. Bisset, V. S. A. Kumar, and M. V. Marathe, "EpiNet: A simulation framework to study the spread of malware in wireless networks," in *2nd International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, (SimuTools) 2009*, O. Dalle, G. A. Wainer, L. F. Perrone, and G. Stea, Eds. ICST Press, Mar. 2–6 2009, p. 6, rome, Italy.

[4] K. Wendelsdorf, J. Bassaganya-Riera, R. Hontecillas, and S. Eubank, "Model of colonic inflammation: Immune modulatory mechanisms in inflammatory bowel disease," *Journal of Theoretical Biology*, 2010.

[5] S. A. Manavski and G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment," *BMC Bioinformatics*, 2008.

[6] L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," *GPU Gems*, vol. 3, pp. 677–695, 2007.

[7] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating Molecular Modeling Applications with Graphics Processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618–2640, 2007.

[8] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. V. Marathe, "Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks," in *SC '08: ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[9] S. Eubank, H. Guclu, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wan, "Modelling disease outbreaks in realistic urban social networks," *Nature*, vol. 429, pp. 180–184, 2004.

[10] R. Blendon, L. Koonin, J. Benson, M. Cetron, W. Pollard, E. Mitchell, K. Weldon, and M. Herrmann, "Public response to community mitigation measures for pandemic influenza," *Emerging Infect. Diseases*, vol. 14, no. 5, 2008.

[11] J. M. Epstein, "Modelling to contain pandemics," *Nature*, vol. 460, no. 7256, p. 687, 2009.

[12] R. J. Beckman, K. A. Baggerly, and M. D. McKay, "Creating synthetic base-line populations," *Transportation Research A – Policy and Practice*, vol. 30, pp. 415–429, 1996.

[13] C. Barrett, R. Beckman, K. Berkbigler, K. Bisset, B. Bush, K. Campbell, S. Eubank, K. Henson, J. Hurford, D. Kubicek, M. Marathe, P. Romero, J. Smith, L. Smith, P. Speckman, P. Stretz, G. Thayer, E. Eeckhout, and M. Williams, "TRANSIMS: Transportation Analysis Simulation System," LANL, Tech. Rep. LA-UR-00-1725, 2001.

[14] "Census of Population and Housing," US Census Bureau. http://www.census.gov/ prod/cen2000/, Tech. Rep., 2000.

[15] "National Household Transportation Survey," US Department of Transportation, Federal Highway Administration. http://nhts.ornl.gov/, Tech. Rep., 2001.

[16] S. Eubank, C. Barrett, R. Beckman, K. Bisset, L. Durbeck, C. Kuhlman, B. Lewis, A. Marathe, M. Marathe, and P. Stretz, "Detail in network models of epidemiology: are we there yet?" *Journal of Biological Dynamics*, vol. 4, no. 5, pp. 446–455, 2010.

[17] NVIDIA, "NVIDIA CUDA Programming Guide-3.0," 2010, http://developer.download.nvidia.com/compute/cuda/3_0/ toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

[18] A. M. Aji, K. R. Bisset, and W. Feng, "Modeling kernel offload methodologies in multi-gpu environments: A case study," NDSSL, Tech. Rep., 2010.