HHS Public Access

Author manuscript

IEEE Int Symp Parallel Distrib Process Workshops Phd Forum. Author manuscript; available in PMC 2016 August 17.

Published in final edited form as:

IEEE Int Symp Parallel Distrib Process Workshops Phd Forum. 2016 May ; 2016: 463–472. doi:10.1109/IPDPSW.2016.20.

Real-Time Agent-Based Modeling Simulation with in-situ Visualization of Complex Biological Systems:

A Case Study on Vocal Fold Inflammation and Healing

Nuttiiya Seekhao[*], **Caroline Shung**[†], **Joseph JaJa**[*], **Luc Mongeau**[†], and **Nicole Y. K. Li-Jessen**[‡]

[*]Department of Electrical and Computer Engineering - University of Maryland-College Park, Maryland, USA

[†]Department of Mechanical Engineering - McGill University, Montreal, Canada

[‡]School of Communication Sciences and Disorders - McGill University, Montreal, Canada

Abstract

We present an efficient and scalable scheme for implementing agent-based modeling (ABM) simulation with In Situ visualization of large complex systems on heterogeneous computing platforms. The scheme is designed to make optimal use of the resources available on a heterogeneous platform consisting of a multicore CPU and a GPU, resulting in minimal to no resource idle time. Furthermore, the scheme was implemented under a client-server paradigm that enables remote users to visualize and analyze simulation data as it is being generated at each time step of the model. Performance of a simulation case study of vocal fold inflammation and wound healing with 3.8 million agents shows 35× and 7× speedup in execution time over single-core and multi-core CPU respectively. Each iteration of the model took less than 200 ms to simulate, visualize and send the results to the client. This enables users to monitor the simulation in real-time and modify its course as needed.

Keywords

component; HPC; ABMs; In Situ Visualization; Heterogeneous Platform; Systems Biology

I. Introduction

Agent-based modeling (ABM) is a powerful and widely used approach to quantitatively simulate a system defined by a set of autonomous agents that operate and interact in discrete time steps. ABMs represent models at the microscale, which attempt to explain the emergence of higher order properties of the overall system. Depending on the system being modeled, each *agent* can represent a wide variety of entity types in an environment ranging from living cells in a biological process modeling, animals in an ecosystem modeling, to cities or countries in an economic model. These agents 'live' in their environment, or *world*,

Contact: nseekhao@umiacs.umd.edu.

whose organization may vary substantially depending on the particular application. In this work, we constrain the world to a two-dimensional grid whose size is determined by the granularity of the simulation. For complex biological systems such as inflammatory and wound healing response, the world consists of a grid of tissue patches, each patch may contain a number of entities such as cells and extra-cellular matrix (ECM) proteins. The size of the grid reflects the granularity of the simulation, and hence the larger the grid the more accurate the simulation. However, high fidelity simulation typically introduces a significant computational burden that, when coupled with the work needed to perform in-situ visualization, makes the overall task of real-time simulation and visualization quite challenging. Thus, such high fidelity simulations stand to benefit substantially from an efficient and scalable parallel implementation.

A challenge in biological simulation is to handle the differences in spatiotemporal scales between cellular and chemical interactions [1]. For example, cellular movements occur at the rate of micrometers per hour ($\mu m/\tau$), while cytokine diffusion in tissue occurs at the rate of micrometers per second ($\mu m/t$). A naive approach would be to simulate the model at the smallest temporal scale required, i.e. time step $ts = t$. Clearly, this would unnecessarily increase the complexity of the coarse-grain processes. To solve this problem, we design a mechanism that captures the behavior of the finer-scale processes over a coarse time window using convolution, and offload this intensive computation to the GPU while the CPU cores focus on coarse-grain processes.

Visualization is a crucial component of any ABM simulation and is usually done separately on the stored data that was generated during the simulation. To date, most visualization techniques proposed fall into one of the following categories; local simulation, conventional work-flow remote simulation [2], or *client-render remote simulation*. In *local simulation*, the visualization happens in the same place where the computation is performed. Thus, this assumes a monitor attached *locally* to the computing platform, which means that, in order to take advantage of a powerful server, the user needs to have a physical access to it. This solution is not acceptable since servers are usually maintained in an isolated highly-regulated area, which is only accessible to the users via a secured network protocol. A commonly used model, *conventional work-flow remote simulation*, performs computational part of the simulation on the server first, store data on disk for later visualization on the client machine. This approach requires temporary storage and heavy traffic on disk. Note that this approach precludes computation steering. The last category is rarely seen, but is mentioned here for completeness, namely the *client-render remote simulation*. This scheme performs the simulation on the server then send the rendering commands to the client. This leaves all rendering responsibility to the client's local computing resource, which is usually much less powerful than that of the server. Existing well-known ABM platforms use a mix of strategies for visualization. NetLogo assumes local simulation [3], while SPADES uses conventional work flow [4]. MASON and FLAME GPU allow for both conventional work flow and computation/visualization coupling [5], [6]. No server-client rendering protocol, however, were specified for the latter option, thus it is fair to assume the local simulation model was used for the coupling of computation and visualization.

In situ visualization, or in-place simulation output processing, addresses all the issues other visualization work flows pose. A quadtree-based ABM is proposed by [7] to reduce the amount of irrelevant data analyzed in-situ, where [8] attempts to accomplish the same goal with a bitmap-based approach. Paraview Catalyst [9], [10] was developed to process simulation output data in-situ according to the user's co-processing script. An image-based approach built on top of Paraview Catalyst was presented by [11] to efficiently manage rendered images created in-situ by Paraview Catalyst. As much as all these work ([7]-[11]) reduce I/O loads, none completely by-pass I/O.

In the present study, VirtualGL is used in the implementation, resulting in an In Situ visualization ABMs framework that completely by-passes the disk as a mediator in the visualization pipeline. Our main goal is to be able, for each time step of the model, to perform the simulation and visualization in a few hundred milliseconds, including the transfer time of the visualization and statistical summary information to the remote client. Such a performance enables the users to take full advantage of the computational power of the server, while analyzing and steering the computation in real-time.

II. Overview and Background

A. Heterogeneous Computing Platform

Heterogeneous computing systems refer to a diverse set of computing resources interconnected via high speed network to collaboratively support execution of computationally intensive parallel and distributed applications [12]. Heterogeneous platforms of various architectures and scales are quite popular. For example the larger scale platforms are based on large clusters of different types of multicore CPUs and many-core accelerators such as GPUs. In fact, almost all current personal computers are based on heterogeneous computing platforms that include a multicore CPU with an attached accelerator of one or more GPUs. However, most often the applications do not make effective use of these available resources. For example, if the CPU is only there to move data and launch GPU kernels, or the GPU is there to merely act as an accelerator to the CPUs, the program is not really employing the full power of the heterogeneous computing environment. On the other hand, if both CPUs and GPUs collaborate to handle important computations, then major performance gains are possible. But this requires a careful scheduling and orchestration of the operations using the available resources. In this work, we will focus on a single node platform consisting of a multi-core CPU with one or several many-core GPUs attached to it.

1) Multi-Core Central Processing Units (CPUs)—Driven by a performance hungry market, there is always a demand for faster processor regardless of the speed of the fastest available processor at the time. Moore's law predicts that the number of transistors in a chip doubles every 18 months [13]. And continuous performance improvement of a processor has been relying on increase in density of integrated circuits (ICs) on a chip for decades [14], [15]. However, according to Pollack's rule, performance increase by microarchitecture alone is roughly proportional to square root of increase in complexity [16], thus the performance of a single processor core does not scale linearly with the number of logic on the core. As

the transistor size shrinks, the leakage current becomes larger [17]. And with higher integrated density, power dissipation becomes the bottleneck of the architecture [16], [17]. Alternatively, performance boost could be achieved by increasing the clock speed, or the frequency at which the processor operates at. This gives more instructions per second, however, due to increased dynamic power dissipation and design complexity, the clock frequency is currently limited to about 4 GHz [18]. Multicore architecture allows scalable processor design and offers a way to achieve better performance without infringing the power dissipation requirements [16]-[18].

Today, a CPU chip typically consists of 2 to 10 CPU cores. A powerful compute node may consist of multiple CPU sockets resulting in more number of cores, typically 16 to 20. For more computing power, multiple compute nodes can work together in a cluster and communicate among themselves via high-speed connections.

2) Graphics Processing Units (GPUs)—GPUs were originally designed as special purpose processors focusing on graphics computations such as polygon calculations, or image filtering. Since the introduction of the CUDA high level programming environment by NVIDIA, GPUs have become the preferred high performance computing platform especially for data parallel computations, achieving a much better performance/energy tradeoff than multicore CPUs. In general, a GPU consists of thousands of processing cores, making them very suitable for data parallel operations. The scientific community has picked up interest in GPU computing due to their computationally demanding applications, which has given rise to General Purpose GPU (GPGPU). CUDA (II-B) was then introduced in 2007 to enable GPGPU programming in C language with C-like extensions. Since its introduction, more than 100 million computers with CUDA-capable GPUs have been shipped to end users [19].

GPUs consist of a number of Streaming Multiprocessors (SMs), each of which contains a number of Streaming Processors (SPs or cores). The GPUs are capable of launching thousands of threads simultaneously. All the SMs have access to the high bandwidth Device memory (peak bandwidth 240 GB/s). The best bandwidth is achieved when all threads in warp access coalesced memory. In this work, the computation component was tested on a compute node with the Tesla K20c, whereas the whole suite (computation and visualization) was tested on a node with a Tesla K80 GPU. The overview of their architecture is summarized in table I.

B. Programming Environment

Designing with speed and efficiency in mind, a light-weight object-oriented programming language C++ is chosen. To take advantage of multiple CPU cores, the code was extended with Open Multi-Processing (OpenMP) to employ concurrency. OpenMP is a highly portable application programming interface (API) which supports parallel executions on shared-memory platforms via a set of platform-independent compiler directives [20].

To communicate and issue instructions to GPUs, Compute Unified Device Architecture (CUDA) programming interface is used. CUDA is a parallel computing platform and

programming model, which allows general-purpose programming of the GPU via C-like language extension keywords [2]. CUDA assumes a GPU attached to the host (CPU) which control data movement to/from GPU, and is responsible for launching kernels, functions to be executed by all threads launched on the GPU.

Visualization was implemented using Open Graphics Library (OpenGL). OpenGL is an open standard, cross-language API for 2D and 3D rendering. OpenGL is widely used in extensive range of graphics applications for its portability and speed.

C. Agent-Based Modeling (ABM)

Agent-based modeling (ABM) is a powerful bottom-up approach for modeling systems with interacting components to observe emerging behavior and insightful information about the system [21]. The basic components of ABMs are:

- ***Agents*** - Autonomous objects which perform actions and interact with other agents and the environment
- ***Agent Rules*** - Behaviors of each type of agents
- ***World*** - The environment in which all agents 'live' in Multiple types of agents can be modeled in a single ABM.

Agents are usually object instances, thus most ABMs are implemented using object-oriented programs such as C++, or JAVA.

Each type of agents behaves according to a set of pre-defined rules, which can be deterministic or stochastic. For example, a simulation related to tissue inflammation may have various cell types, such as neutrophils, macrophages and fibroblasts, as agents. Rules are then determined using the best available knowledge in literature about the behavior of cells. The autonomous agents are mobile and make decisions based on their states and the world environment. The *world* in our case is modeled as a grid of tiny squares (2D) called *patches*. Patch size is uniform across the world, and thus the resolution of the simulation environment is inversely proportional to patch size.

The temporal dimension of ABMs is discrete and the simulation progresses in sequence of synchronous iterations (sometimes referred to as *tick*). Thus, even if the semantics of agent execution in ABMs is parallel in nature, constant updates and synchronizations at iteration-granularity are inevitable, making the task of designing an efficient parallel algorithm for ABMs challenging.

D. Modeling of Inflammatory and Healing Process in Vocal Folds

Human vocal folds experience continuous biomechanical stresses during phonation. Excessive vocalization can cause phonotrauma, which, like any other forms of mechanical trauma, triggers a highly complex process of inflammation and tissue repair. Treatment outcomes often depend on the level of the initial damage and influenced by individual's genetics or pre-morbid tissue status [22]. Thus, personalized treatments based on individual's biological profile can increase the chance of better healing results. A vocal fold

ABM has been developed to simulate inflammation and repair to gain a deeper mechanistic understanding of the underlying cellular and molecular processes, which has shed insights of rational therapeutic design. Vocal fold wound healing modeling is thus an excellent candidate application to test and validate our proposed parallelization of ABMs, due to its complexity and the availability of patient-specific data [23], [24].

Table II summarizes actions of each agent type for our application. The cells, which includes Platelets, Neutrophils, Macrophages and Fibroblasts, are mobile agents that make action decisions based on the states of their surroundings. At the time of acute injury, the traumatized mucosal tissue within the damaged area triggers platelet degranulation [23], [25]. Different chemokine gradients were readily created and stimulate vasodilation and attraction of inflammatory cells, namely, neutrophils and macrophages. Activated neutrophils and macrophages at the wound site secrete more chemokines to attract fibroblasts and clean up cell debris. Fibroblasts activated by tissue damage deposit extracellular matrix (ECM) proteins such as collagen, elastin, and hyaluronans at the wound area of repair. These ECM proteins then form a scaffold for supporting fibroblasts in wound contraction and other cells' migration and wound repair activities [26]. The flow diagram of the interactions between all the components in the model is shown in Figure 1.

To achieve the best resolution in the ABM world, each patch is made to be the smallest possible for a single cell to occupy. This results in patch size of 15μm × 15 μm [27], [28]. Initial density of cells and ECM proteins were calculated based on empirical data from literature [29]-[33]. The configuration details, which were determined based on our best knowledge of vocal folds anatomy, are shown in table III.

E. Chemical Diffusion

Chemical diffusion is one of the most crucial and highly intensive computational components of the model. Diffusion equation with decay in 2D can be written as follows:

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right) - \gamma c, \quad (1)$$

where c is the chemical concentration, D is the diffusion coefficient and γ is the decay constant. By using a Taylor expansion to discretize the continuous diffusion equation, we get:

$$\begin{aligned} & c(x,y,t+\Delta t) \\ & \quad = c(x,y,t) \\ & \quad + D\Delta t\left[\left(\frac{c(x+\Delta x,y,t)-2c(x,y,t)+c(x-\Delta x,y,t)}{\Delta x^2}\right)+\left(\frac{c(x,y+\Delta y,t)-2c(x,y,t)+c(x,y-\Delta y,t)}{\Delta y^2}\right)\right] \\ & \quad - \gamma\Delta t c(x,y,t) \end{aligned}$$

(2)

In our case, $\Delta x = \Delta y$, thus Eqn. (2) becomes:

$$\begin{aligned} & c(x,y,t+\Delta t) \\ & \quad = \left(1-\frac{4D\Delta t}{\Delta x^2}-\gamma\Delta t\right)c(x,y,t) \\ & \quad + \frac{D\Delta t}{\Delta x^2}\left[c(x+\Delta x,y,t)+c(x-\Delta x,y,t)+c(x,y+\Delta y,t)+c(x,y-\Delta y,t)\right] \end{aligned} \quad (3)$$

Notice that Eqn. (3) is a discrete function that can be implemented easily as a function. However, we need to first make sure that the solution is stable. Using Von Neumann Stability Analysis method to study the growth of the waves e^{ikx} [34], we have the following stability conditions:

$$\frac{D\Delta t}{\Delta x^2}+\frac{D\Delta t}{\Delta y^2} \leq \frac{1}{2} \quad (4)$$

Since $\Delta x = \Delta y$, we have,

$$\Delta t \leq \frac{\Delta x^2}{4D} \quad (5)$$

Given that the largest values of D in our set of chemical types is $900\frac{\mu m^2}{\text{minute}}$, with patch width $\Delta x = 15\mu m$, $\Delta t \leq 0.0625$ *minute*. Clearly, the work complexity of the simulation would be unnecessarily high if we simulate the model at $\Delta\tau = 0.0625$ *minute* rather than $\Delta\tau = 30$ *minutes*.

Fortunately, there is a way to capture Eqn. (3) at a larger time step. By letting $\lambda = \frac{D\Delta t}{\Delta x^2}$, Eqn. (3) can be rewritten as follows:

$$c(x,y,t+\Delta t) = (1-4\lambda-\gamma\Delta t)\cdot c(x,y,t)\ \lambda\cdot c(x+\Delta x,y,t)+\lambda\cdot c(x-\Delta x,y,t)+\lambda\cdot c(x,y+\Delta y,t)+\lambda\cdot c(x,y-\Delta y,t) \quad (6)$$

or,

$$c(x,y,t+\Delta t) = \sum_{l=x-1}^{x+1}\sum_{k=y-1}^{y+1} c(l,k,t)\cdot f(x-l,y-k), \quad (7)$$

where

$$f(x,y) = \begin{cases} 1-4\lambda-\gamma\Delta t & x=0, y=0 \\ \lambda & x=\pm 1, y=\pm 1 \\ 0 & \text{otherwise} \end{cases}$$

Clearly, Eqn. (7) is equivalent to saying $c(x, y, t + \Delta t) = c(x, y, t) * f(x)$, where * represents convolution. Thus, we can fast forward this process to capture diffusion at large time step, $\Delta\tau$, without violating stability constraints using convolution.

In order to compute $c(x, y, \tau + \Delta\tau)$, where $\Delta\tau = m \cdot \Delta t$, we convolve the chemical concentrations from previous step, $c(x, y, \tau)$, with $f(x, y)$, m times. By commutative property of convolution, we can convolve $f(x, y)$ with itself m times to get $f_m(x, y)$, and compute the diffused concentrations at each tick as follows:

$$c(x,y,\tau+\Delta\tau) = c(x,y,\tau) * f_m(x,y) \quad (8)$$

For example, the effective diffusitivity of IL-1β in tissue is $900\frac{\mu m^2}{min}$ [35]. In our 15-μm patch world, simulating at 30-minute time steps, the program has to calculates $c(x, y, \tau) * f_{480}(x, y)$ at each time step. This means a chemical on a given patch (x,y) can diffuse to all patches within $x \pm 480$ and $y \pm 480$, which is a window of dimension 961×961 or approximately 1 million patches.

After obtaining the formula for fast-forward diffusion calculations, we need to also consider boundary conditions to appropriately pad the data for convolution operations. Depending on the area of interest, the padding chosen could either be *constant padding* or *mirror padding* or both.

In our case study of vocal fold modeling, our tissue area of interest has epithelium on the outermost layer. Since the dynamics of vocal fold epithelium is abstracted in this ABM, we

have effectively one wall, or 1-side 0-flux boundaries. And the rest of the walls are padded with empirically obtained baseline chemical levels, or constant padding.

III. Methodology

A. Scheduling and Coordination of the CPU and GPU Computations

As discussed in section II-E, chemical concentration in a patch can affect other patches within a radius of up to 480 patches. In other words, our convolution kernel can be as large as 961 × 961, defining a window that contains roughly a million patches. Fortunately, GPUs are much faster at computing convolutions than CPUs [36]. However, the diffusion needs to be updated at every iteration, which means we need to move data between the CPU and GPU at the end of each iteration, which makes the total time to compute diffusion and move the results back quite significant.

To address this issue, we hide the diffusion computation time by utilizing our GPU and p CPU cores as follows:

- **i)** We allocate $p - 1$ CPU threads for executing parallel operations other than diffusion.
- **ii)** The remaining CPU thread prepares and manages data movement to and from the GPU
- **iii)** GPU computes chemical diffusion using FFT-based convolutions concurrently with the CPU threads executing their operations

Since all agent decisions during time step t are determined by the state of the environment determined at the end of time step $t - 1$, steps (i) and (iii) can be executed simultaneously as shown in Figure 2.

This approach is applicable to chemical diffusion in biological systems such as hormones in the endocrine system, or pharmacokinetics of drug infusions. Furthermore, particle diffusion is encountered in a wide range of system modeling applications. Hence, the technique discussed can be applied to a broad range of system modeling applications involving any type of particle diffusion. The diffusion equation (Eqn. 1) is of the same form as the Heat Equation, which has an even larger range of applications such as the aforementioned particle diffusion, Brownian motion, Schrodinger equation for a free particle, thermal diffusivity, financial mathematics etc. And more importantly, if we generalized this idea, the computational overlap technique discussed can also be applied to any system modeling application with the following properties:

- **1)** Simulation is carried out in *discretized synchronous* temporal steps.
- **2)** All operations in time step t depend solely on the state of the environment and agents determined by the end of time step $t - 1$.
- **3)** Computations in each time step can be divided into multiple independent tasks with at least one task in each of the following categories:

(a) CPU suitable task

(b) Intensive GPU suitable task

If all the three properties above hold, the CPU-GPU computation overlap technique can be applied to any systemmodeling implementation on a heterogeneous (CPU/GPU) computing platform to ensure data and task parallelization, resulting in efficiency and performance improvements on the model computation.

B. Update and Synchronization

All agents make decisions in time step t based on the state of the system in time step $t-1$. Each agent then modifies its own state in time step t according to the decisions made in the same time step. This means there is little to no data dependency in the process of state update for each agent for any specific time step, making ABM simulation an excellent candidate for parallelization. However, as discussed earlier in section II-C, constant updates are unavoidable causing heavy traffic to memory. In order to mitigate memory bandwidth contention, each agent maintains *dirty* flags, and each agent only updates when necessary.

Apart from writing to their own fields, agents' decisions also affect their environment. If we assume that the resolution of the world is the finest possible, each patch will only allow one agent. When an agent targets a patch to move into, it needs to make sure no other agents will move into that patch. Naively, one could maintain a lock for each patch; however locks incur a high overhead. For optimal performance, atomic operations were used for synchronization to enforce the rule of one agent per patch as shown in algorithm 1. The function *atomic_test_and_set(v)* atomically sets the content of v to *true* and return the previous value of v, thus an agent can find out if the patch is available by checking the return value.

C. In Situ Remote Visualization

In recent years, as the computational power of computing platforms has substantially increased, numerical simulations developed on these platforms have grown much more complex, generating outputs that measure up to hundreds of terabytes in sizes (and soon exabytes) [2]. Conventional visualization work-flow of writing output to disk for later visualization is not really a cost effective solution for such cases. To design a simulation framework that scales with the computational power of the latest platforms, a compute-visualize paradigm satisfying the following properties will be extremely desirable.

1) Both the computation and the visualization will take full advantage of computational power of the server;

2) The load on the disks should be minimized;

3) The researcher should be able to steer the computation based on the data as it is being generated and visualized.

The *local simulation*, as previously discussed, can rarely take advantage of the server as it assumes direct connection between the compute node and the display and it is uncommon for the user to have physical access to the server. On the other hand, *conventional work-flow remote simulation* may be able to partly take advantage of the powerful server, but it doesn't

exhibit neither property 2 nor 3. Lastly, the *client-render remote simulation* scheme may manifest both 2 and 3; however, it doesn't fully take advantage of the powerful server since it redirects the rendering to the client.

The In Situ Remote Visualization paradigm [37] exhibits all three of the desired properties. Our ABM implementation was developed and tested on a system configured with VirtualGL and TurboVNC. VirtualGL is an open source package which gives any Unix or Linux remote display software the ability to run OpenGL applications with full 3D hardware acceleration [38]. Figure 4 shows the X11 transport with an X proxy diagram. The application uses Xlib to communicate with the 3D X server to request for an openGL context. Once the context is created, the application can then talk directly to the rendering hardware via libGL. An X proxy, in our case TurboVNC, essentially acts as a virtual X server. The X11 rendering is then performed to a virtual framebuffer in main memory rather than a real framebuffer on the graphics card. This allows the X proxy to compress and transmit the buffer content to end user without the need to provide any X server capabilities, thus a very thin client can be used.

Once the remote visualization protocol has been established, the next step is to make sure the rendering code is efficient. The visualization code is optimized using Vertex Buffer Objects (VBOs) as well as Index Buffer Objects (IBOs), which is a way for OpenGL to reserve fast graphics memory. ABMs generally consist of at least one plane, the *world* plane. The world plane is usually heterogeneous in patch type. In the case that the programmer knows that the ratio of a certain type of patches to other types is high, he or she can make that patch type a *base* type, spread them across the world using one big texture and store the coordinates of their outline in a VBO, and render other types of patches on top. This strategy reduces the number of OpengGL draw calls to the number of patches that are not of the *base* type, which can result in a significant work reduction in many cases. For example, the vocal fold model, about 80% of the patches are tissue, then the rest are capillary and epithelial patches. In this case, the calls to render the world plane can be reduced by 80% using the technique discussed.

IV. Performance

A. Computation Only

Different versions of Vocal Fold ABM were implemented for performance evaluation purposes as shown in table IV. These versions follow the same model rules, but differ in computing resource utilization. They were tested and bench-marked on a compute node with 16-core Intel(R) Xeon(R) E5-2690 CPU and NVIDIA Tesla K20c GPU. As shown in Figure 5, the GPU-mCPU-Overlap implementation achieves the best performance. This implementation follows the techniques discussed in Section III, where the ABM model execution is being divided up into smaller more manageable tasks that are either high-throughput computationally-intensive or complex, but less computationally intensive. The former is considered GPU-suitable, thus is executed on the GPU, whereas the latter gets executed on the CPU. The CPU-suitable tasks are then further sped up by multiple CPU

threads. The total time to execute one iteration of the program is governed by the following equation:

$$t_{\text{total}} = max\left\{t_{CPU_{\text{maxthreads}}}, t_{GPU_{\text{maxthreads}}}\right\} + t_{sync}, \quad (9)$$

where $t_{CPU_{\text{maxthreads}}}$ and $t_{GPU_{\text{maxthreads}}}$ are the time consumed by executing tasks using maximum number of threads on CPU and GPU respectively. The maximum number of threads typically corresponds to the number of physical cores on the specified computing device. t_{sync} is the time it takes to synchronize the data resulting from task executions on CPU and GPU. If $t_{\text{device1}_{\text{maxthreads}}} \geq t_{\text{device2}_{\text{kthreads}}}$, then clearly, any number of threads launched beyond k threads on *device2* would not benefit the overall performance. For the vocal folds simulation on the afore-mentioned compute node, $t_{GPU_{\text{maxthreads}}} \geq t_{CPU_{\text{8threads}}}$, thus the load is most balanced when executing with 8 CPU threads.

Our implementations are able to execute the vocal fold ABM at a scale, which is infeasible on a popular existing ABM framework, NetLogo [3]. To demonstrate the performance gain of the proposed techniques compared to an existing ABM framework, we obtain the performance of the GPU-mCPU-overlap implementation running at a scale feasible on NetLogo. For a 1-million patch world, with half number of initial cells, the model runs on average 36.6 s per iteration on NetLogo and an average of 0.091 s per iteration on the GPU-mCPU-overlap implementation, resulting in a **400×** speedup.

Despite differences in underlying hardware, D'Souza's work on Tuberculosis (TB) ABM Simulation [43] is arguably most suitable for performance comparison with the work reported in this paper. The aforementioned TB ABM describes a complex multi-scale biological system of agents that communicate via chemical signals, which aligns in most respects with our model. The largest case reported in their work consists of 256 patches x 256 patches world with 100 initial Macrophages, and takes 450 seconds to run for a 4-day simulation. In comparison, our case study consists of 30× world size with 1000× the number of initial cells, and takes only 25 seconds, i.e. 20× less, to perform a 4-day simulation.

Next, we compare the performance improvement gained by the GPU-mCPU-overlap implementation over other low-level highly optimized implementations. A 5-day high-resolution simulation that takes 20 minutes on CPU only takes half a minute when both CPU and GPU are efficiently utilized via our proposed task orchestration technique, accounting for a **35.1×** and **6.6×** speedup in execution time over single-core and multi-core CPU respectively. This improvement is significant given the fairly complex and biologically representative 2D model with intensive calculations and heavy memory traffic.

B. Computation + Visualization

We coupled the GPU-mCPU-overlap computation implementation, which shows the best performance from section IV-A with visualization code implemented with OpenGL. The advanced visualization component displays aggregated statistics and simulation state of

multiple components over spatio-temporal dimensions simultaneously. This complex simulation suite (Figure 6, 7) is then tested and benchmarked on a compute node which consists of a 16-core Intel(R) Xeon(R) CPU E5-2630 and an NVIDIA Tesla K80 GPU with rendering enabled. As shown in table VI, average execution time per tick, which includes complex simulation computation and rendering on the server, takes a little bit less than 200 ms. VirtualGL and TurboVNC enable simulation frames to be transmitted to the end user with very minimal overhead. Therefore, the total time from the start of the iteration execution to the time the simulation output frame gets completely rendered on the client terminal can be kept under 200 ms.

V. Conclusion

We presented an efficient ABM task scheduling and management technique which optimally utilizes both multicore CPU and many-core GPU on a single heterogeneous compute node simultaneously. The techniques proposed showed a speedup of **35×** over an optimized sequential implementation when benchmarked with a complex biological modeling application of vocal folds inflammation and wound healing. More importantly, the proposed technique can be generalized to improve efficiency and performance of many complex discrete synchronized time step simulations which can be partitioned into smaller tasks that are either high-throughput and computationally-intensive (GPU-suitable) or more complex but less computationally-intensive (CPU-suitable).

The model computation is then coupled with an advanced visualization component which displays aggregated statistics and simulation state of multiple components over spatio-temporal dimensions. To take full advantage of the powerful computational server, minimize disk load, and enable computational steering, the program was tested and benchmarked on the system with X11 transport via X proxy protocol configured. In-situ visualization along with optimization using OpenGL buffer objects and base-type main plane bring the total time to under 200 ms per iteration enabling remote real-time simulation and visualization.

VI. Future Work

While the proposed techniques resulted in a significant improvement on efficiency and speedup of a fairly complex ABM simulation, there is still room for further optimization. Future work includes optimization of GPU implementation on multi-device GPU chips (2D) and high performance clusters for the 3D case. Changes in data structures to improve spatial locality and memory access are being explored. Additional visualization functionalities such as computational steering input user interface are being expanded to aid users in obtaining more insightful information from the simulation.

Acknowledgment

The authors would like to thank Yun (Yvonna) Li and Alireza Najafi Yazdi for their contributions to the development of the initial model. Sujal Bista for guidance in developing the visualization component. And UMIACS staffs for assistance in VirtualGL and TurboVNC configuration. Research reported in this publication was supported by National Institute of Deafness and other Communication Disorder of the National Institutes of Health under award number R03DC012112 and R01DC005788.

References

[1]. Cilfone NA, Kirschner DE, Linderman JJ. Strategies for efficient numerical implementation of hybrid multiscale agent-based models to describe biological systems. Cellular and Molecular Bioengineering. 2014; 8(1):119–136. [PubMed: 26366228]

[2]. Nvidia, C. Remote visualization on server-class tesla gpus. 2007.

[3]. Tisue, S.; Wilensky, U. International conference on complex systems. Boston, MA: 2004. Netlogo: A simple environment for modeling complexity; p. 16-21.

[4]. Riley, PF.; Riley, GF. Next generation modeling iii-agents: Spades—a distributed agent simulation environment with software-in-the-loop execution. Proceedings of the 35th conference on Winter simulation: driving innovation; Winter Simulation Conference; 2003; p. 817-825.

[5]. Luke S, Cioffi-Revilla C, Panait L, Sullivan K. Mason: A new multi-agent simulation toolkit. Proceedings of the 2004 swarmfest workshop. 2004; 8:44.

[6]. Richmond P, Walker D, Coakley S, Romano D. High performance cellular level agent-based simulation with flame for the gpu. Briefings in bioinformatics. 2010; 11(3):334–347. [PubMed: 20123941]

[7]. Krekhov, A.; Grüninger, J.; Schlönvoigt, R.; Krüger, J. SIGGRAPH Asia 2015 Visualization in High Performance Computing. ACM; 2015. Towards in situ visualization of extreme-scale, agent-based, worldwide disease-spreading simulations; p. 7

[8]. Su, Y.; Wang, Y.; Agrawal, G. In-situ bitmaps generation and efficient data analysis based on bitmaps; Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing; ACM. 2015; p. 61-72.

[9]. Bauer, AC.; Geveci, B.; Schroeder, W. The paraview catalyst user's guide. 2013.

[10]. Ayachit, U.; Bauer, A.; Geveci, B.; O'Leary, P.; Moreland, K.; Fabian, N.; Mauldin, J. Paraview catalyst: Enabling in situ data analysis and visualization; Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization; ACM. 2015; p. 25-29.

[11]. Ahrens, J.; Jourdain, S.; O'Leary, P.; Patchett, J.; Rogers, DH.; Petersen, M. An image-based approach to extreme scale in situ visualization and analysis; Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; IEEE Press. 2014; p. 424-434.

[12]. Topcuoglu H, Hariri S, Wu M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. Parallel and Distributed Systems, IEEE Transactions on. 2002; 13(3): 260–274.

[13]. Schaller RR. Moore's law: past, present and future. Spectrum, IEEE. 1997; 34(6):52–59.

[14]. Hammond L, Nayfeh BA, Olukotun K. A single-chip multiprocessor. Computer. 1997; (9):79–85.

[15]. Venu, B. arXiv preprint arXiv:1110.3535. 2011. Multi-core processors-an overview.

[16]. Borkar, S. Thousand core chips: a technology perspective; Proceedings of the 44th annual Design Automation Conference; ACM. 2007; p. 746-749.

[17]. Roy, A.; Xu, J.; Chowdhury, MH. Multi-core processors: A new way forward and challenges. Microelectronics, 2008. ICM 2008; International Conference on; IEEE. 2008; p. 454-457.

[18]. Parkhurst, J.; Darringer, J.; Grundmann, B. From single core to multi-core: preparing for a new exponential; Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design; ACM. 2006; p. 67-72.

[19]. Wen-Mei, WH. GPU Computing Gems Emerald Edition. Elsevier; 2011.

[20]. Dagum L, Enon R. Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE. 1998; 5(1):46–55.

[21]. Page, SE. The New Palgrave Dictionary of Economics. Palgrave MacMillan; New York: 2005. Agent based models.

[22]. Li N, Verdolini K, Clermont G, Mi Q, Rubinstein EN, Hebda PA, Vodovotz Y. A patient-specific in silico model of inflammation and healing tested in acute vocal fold injury. PloS one. 2008; 3(7):e2789. [PubMed: 18665229]

[23]. Li NY, Vodovotz Y, Kim KH, Mi Q, Hebda PA, Abbott KV. Biosimulation of acute phonotrauma: an extended model. The Laryngoscope. 2011; 121(11):2418–2428. [PubMed: 22020892]

[24]. Abbott KV, Li NY, Branski RC, Rosen CA, Grillo E, Steinhauer K, Hebda PA. Vocal exercise may attenuate acute vocal fold inflammation. Journal of Voice. 2012; 26(6):814-e1.

[25]. Li NY, Vodovotz Y, Hebda PA, Abbott KV. Biosimulation of inflammation and healing in surgically injured vocal folds. The Annals of otology, rhinology, and laryngology. 2010; 119(6): 412.

[26]. Bainbridge, P., et al. Wound healing and the role of fibroblasts. 2013.

[27]. Bettega D, Calzolari P, Doglia SM, Dulio B, Tallone L, Villa AM. Technical report: cell thickness measurements by confocal fluorescence microscopy on c3h10t1/2 and v79 cells. International journal of radiation biology. 1998; 74(3):397–403. [PubMed: 9737542]

[28]. F., RA, Jr.. Nanomedicine, Volume I: Basic Capabilities 8.5.1 cytometrics. 1999. http://www.nanomedicine.com/NMI/8.5.1.htm

[29]. Catten M, Gray SD, Hammond TH, Zhou R, Hammond E. Analysis of cellular location and concentration in vocal fold lamina propria. Otolaryngology-Head and Neck Surgery. 1998; 118(5):663–667. [PubMed: 9591866]

[30]. Hahn MS, Kobler JB, Zeitels SM, Langer R. Midmembranous vocal fold lamina propria proteoglycans across selected species. Annals of Otology, Rhinology & Laryngology. 2005; 114(6):451–462.

[31]. Muñoz-Pinto D, Whittaker P, Hahn MS. Lamina propria cellularity and collagen composition: an integrated assessment of structure in humans. Annals of Otology, Rhinology & Laryngology. 2009; 118(4):299–306.

[32]. Hahn MS, Kobler JB, Starcher BC, Zeitels SM, Langer R. Quantitative and comparative studies of the vocal fold extracellular matrix i: elastic fibers and hyaluronic acid. Annals of Otology, Rhinology & Laryngology. 2006; 115(2):156–164.

[33]. Hahn MS, Kobler JB, Zeitels SM, Langer R. Quantitative and comparative studies of the vocal fold extracellular matrix ii: collagen. Annals of Otology, Rhinology & Laryngology. 2006; 115(3):225–232.

[34]. LeVeque, RJ. Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems. Vol. 98. Siam; 2007.

[35]. Spiros, A. Alzheimer's In Silico diffusion of molecules. Feb. 2000 http://www.math.ubc.ca/~ais/website/status/diffuse.html

[36]. Garland M, LeGrand S, Nickolls J, Anderson J, Hardwick J, Morton S, Phillips E, Zhang Y, Volkov V. Parallel computing experiences with cuda. IEEE micro. 2008; (4):13–27.

[37]. Nvidia, C. Compute unified device architecture programming guide. 2014.

[38]. Project, TV. VirtualGL background. 2015. http://www.virtualgl.org/About/Background

[39]. WDC3D. 6 seamless organic textures 1. Apr. 2010 http://wdc3d.com/blog/textures/6-seamless-organic-textures-1/

[40]. C., I.; van Waveren, JMP. Real-time normal map dxt compression. Feb. 2008 http://www.nvidia.com/object/real-time-normal-map-dxt-compression.html

[41]. Nøttaasen, L. Beach wood texture. Oct. 2009 https://www.flickr.com/photos/magnera/4022717270

[42]. Plain water (seamless) texture high quality. http://textures101.com/view/3551/Plain/Plain_Water_Seamless

[43]. D'Souza, RM.; Lysenko, M.; Marino, S.; Kirschner, D. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units; Proceedings of the 2009 Spring Simulation Multiconference; Society for Computer Simulation International. 2009; p. 21

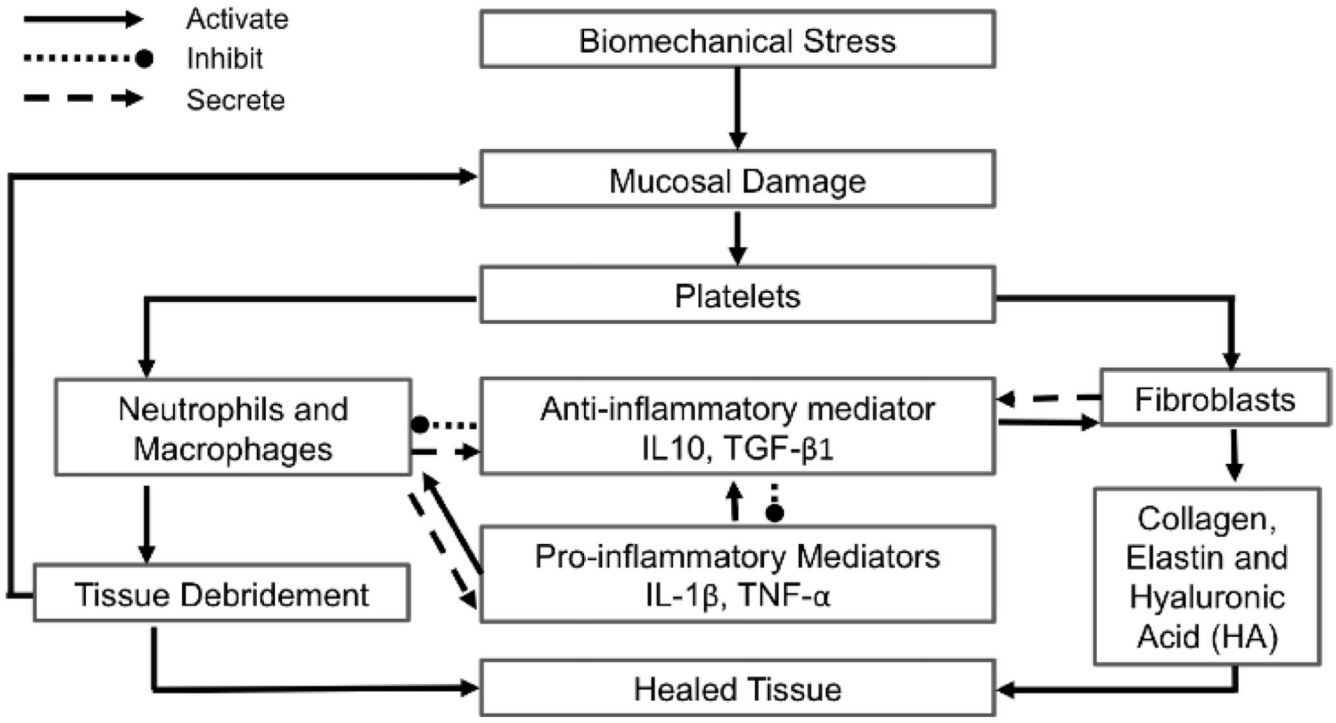

Figure 1.
Flowchart of Vocal Fold ABM. Modified from [22].

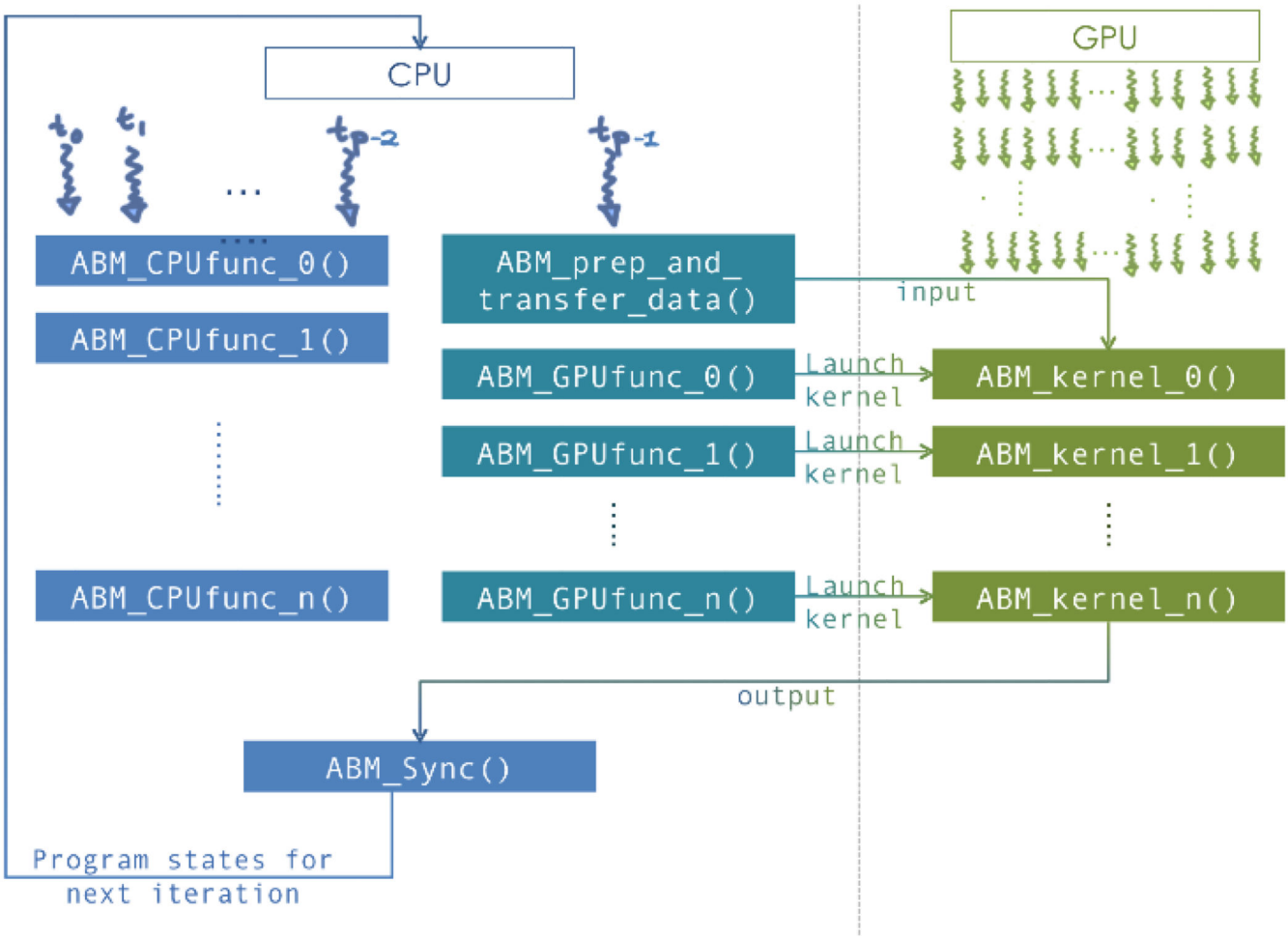

Figure 2.
A diagram depicting the proposed CPU-GPU computation overlap technique (GPU-mCPU-overlap).

Algorithm 1: Agent Atomic Move

Input: $P =$ target patch
Output: $rc = true$ if move was successful, $false$ otherwise

```
if ¬atomic_test_and_set(P.isoccupied) then
    this.x ← P.x;
    this.y ← P.y;
    return true;
else
    return false;
end
```

Figure 3.
Pseudo code demonstrating atomic move algorithm for enforcing one agent per patch rule efficiently.

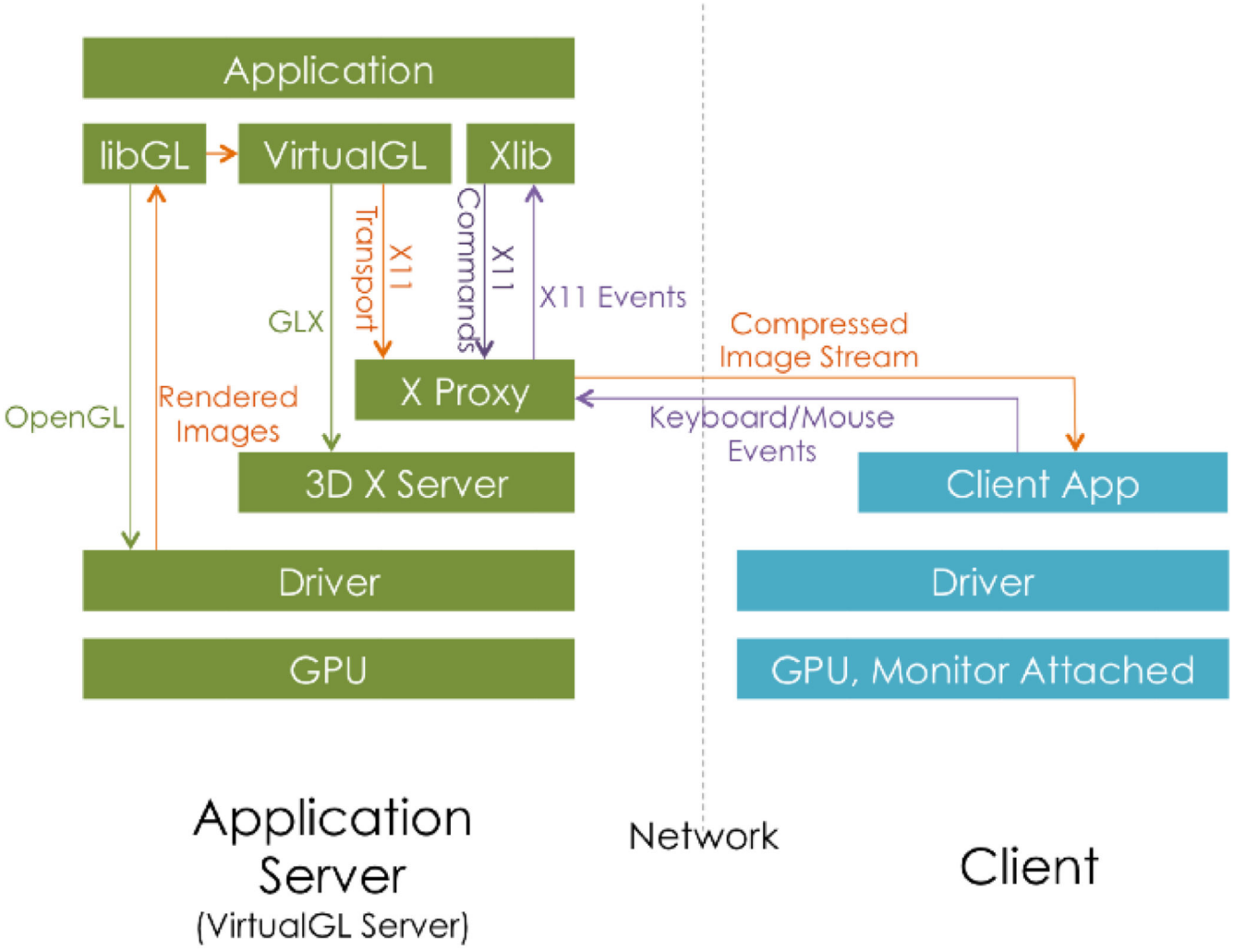

Figure 4.
Diagram depicting the system configuration for In Situ remote visualization using X11 transport with an X proxy.

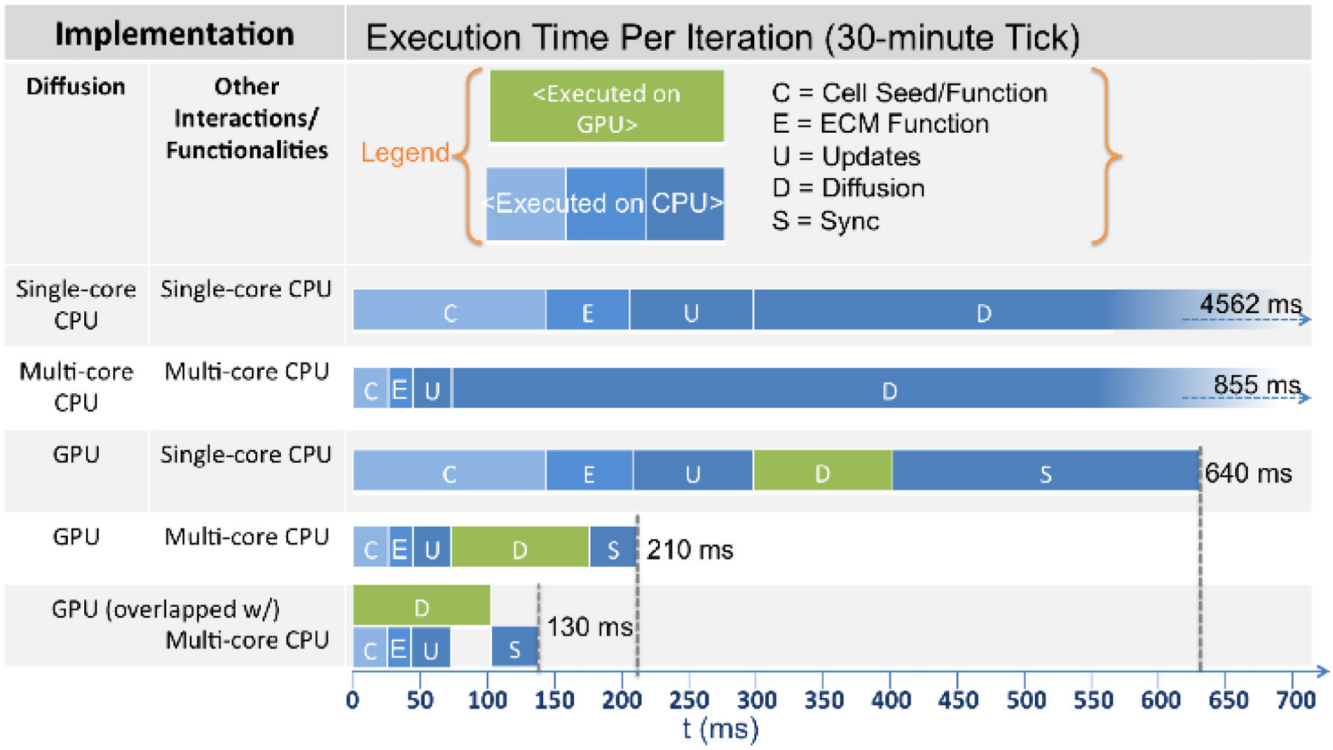

Figure 5.
Performance of 2D Vocal Fold Inflammation and healing ABM on different processing platforms.

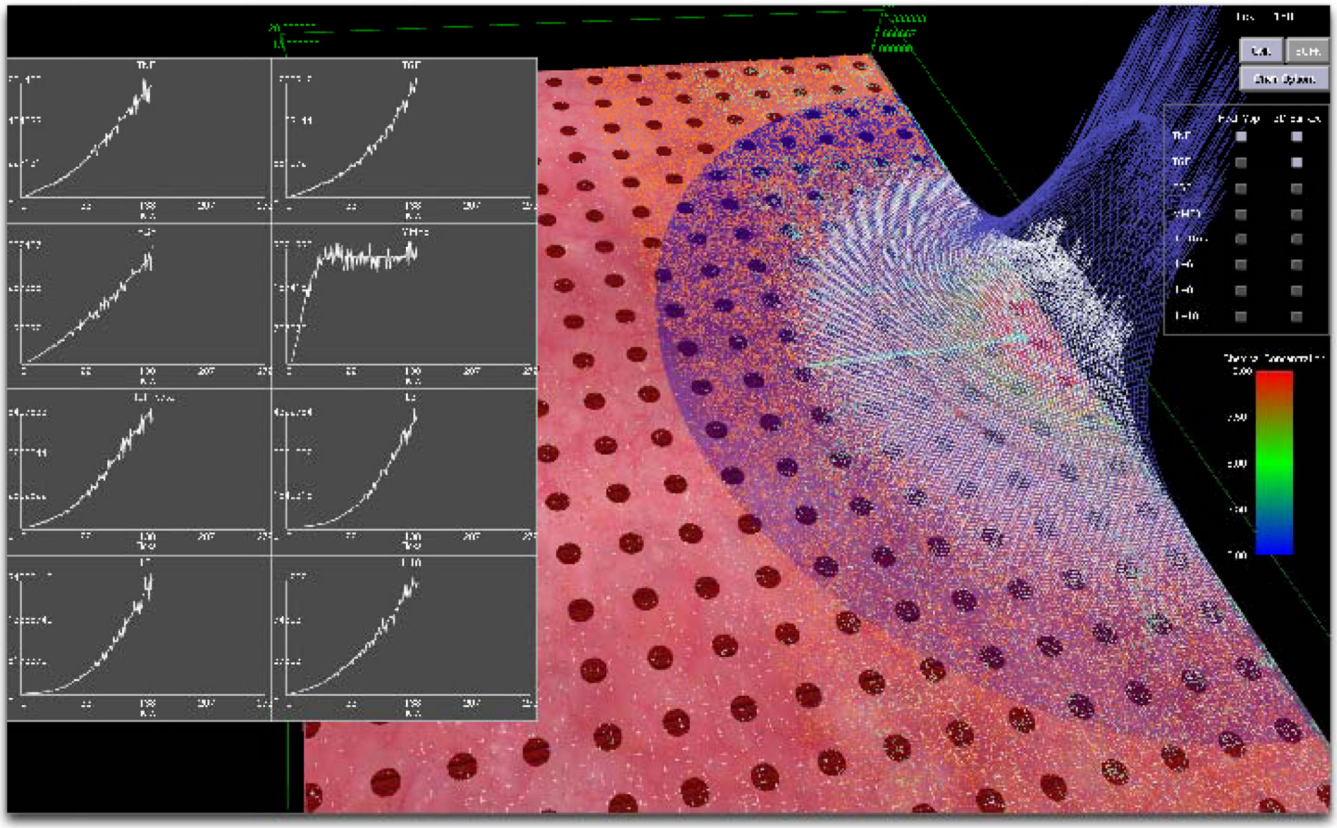

Figure 6.
A screenshot of a running vocal fold inflammatory and wound-healing process with aggregated chemical statistics plots and chemical visualization on (heat map and surface plots)[1].

[1]Texture sources: [39], [40]

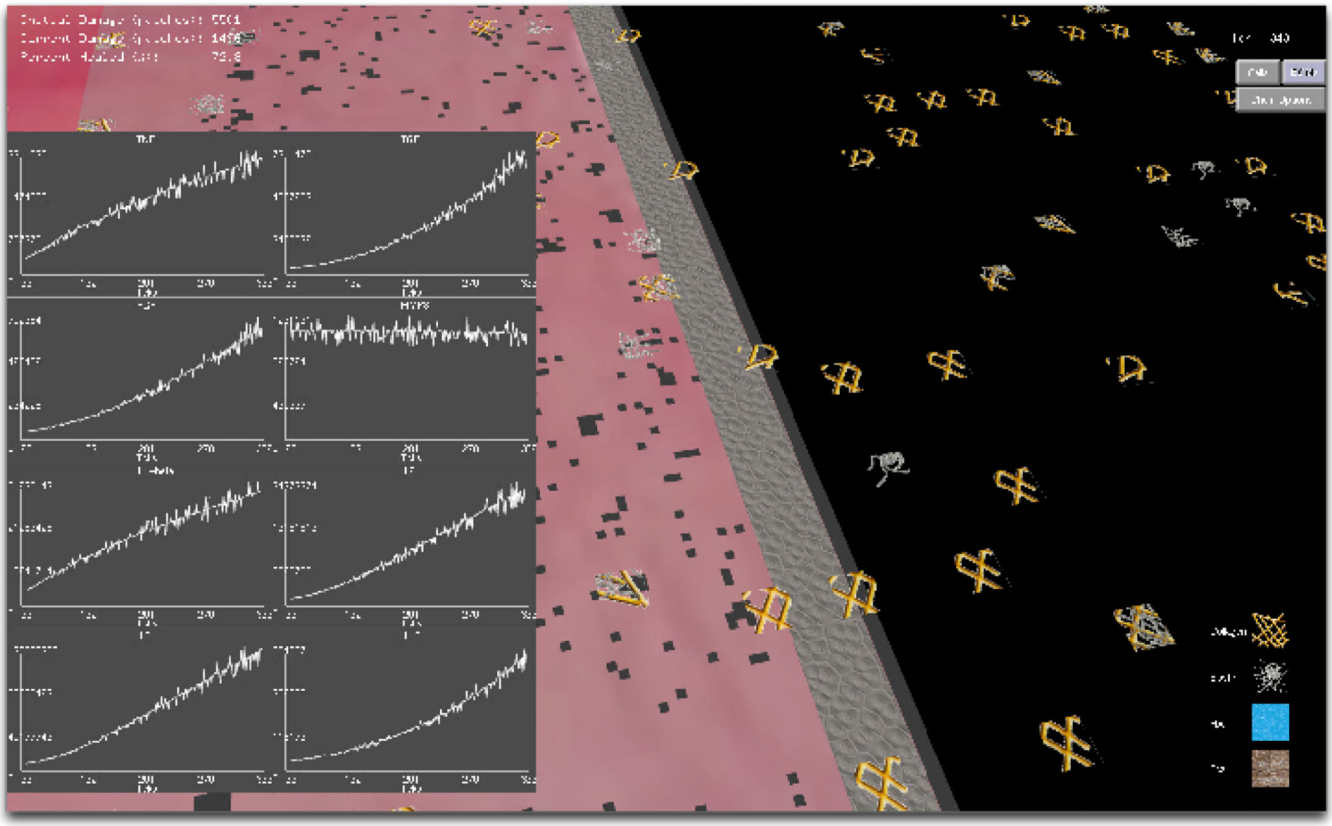

Figure 7.
A screenshot of a running vocal fold inflammatory and wound-healing process with aggregated chemical statistics plots and ECM visualization on[2].

[2]Texture sources: [39]-[42]

Table I

Summary of Tesla C2050 and K20c Specifications

GPU	Tesla K20c	Tesla K80
SMs (per Device)	13	13
CUDA Cores per SM	192	192
Registers per SM	64k	64k
Configurable L1 Cache + Shared Memory per SM	64 kB	128 kB
L2 Cache Size	1.25 MB	1.50 MB
Global Memory (per Device)	4.7 GB	11.25 GB
Max Clock Rate	0.71 GHz	0.82 GHz
Memory Clock Rate	2.6 GHz	2.5 GHz
Memory Bandwidth	208 GB/s	240 GB/s
Compute Capability	3.5	3.7

Table II

Summary of Agent Rules

Agent	Actions
Platelets	Secrete TGF, MMP8 and IL-1,β to attract other cells.
Neutrophils	Secrete TNF and MMP8 to attract other Neutrophils and Macrophages.
Macrophages	Secrete TNF, TGF, FGF, IL-1,β, IL-6, IL-8, IL-10 to attract Neutrophils, other Macrophages and Fibroblasts.
	Clean up cell debris.
Fibroblasts	Secrete TNF, TGF, FGF, IL-6, IL-8 to attract Neutrophils, Macrophages and other Fibroblasts.
	Deposit ECM proteins to repair tissue damage.
ECM Managers	Manages ECM functions and conversion.
	One Manager per patch.

Table III

Summary of Simulation Configurations

Item	Unit	Size
World	patches × patches	1660 × 1160
Patch	μm × μm	15 × 15
	patches	1.9M
Simulated area	mm × mm	24.9 × 17.4
Simulated time-step	minutes	30
Neutrophils	cells	182.4k
Macrophages	cells	22.8k
Fibroblasts	cells	22.8k

Table IV

Implementation Summary

Implementation	Single-core CPU	Tasks Executed on Multi-core CPU	GPU
sCPU-sCPU	Diffusion Other functions	-	-
mCPU-mCPU	-	Diffusion Other functions	-
GPU-sCPU	Other functions	-	Diffusion
GPU-mCPU	-	Other functions	Diffusion
GPU-mCPU-overlap	-	Other functions	Diffusion

Table V

Performance Comparison of Various Implementations

Implementation	Execution Time (ms/tick)	Speedup over Serial Execution
sCPU-sCPU	4562	**1.0×**
mCPU-mCPU	855	**5.3×**
GPU-sCPU	640	**7.1×**
GPU-mCPU	210	**21.7×**
GPU-mCPU-overlap	130	**35.1×**

Table VI

Average Execution Time of Remote In Situ Simulation

	Average Execution Time (ms/tick)
Computation	142
Rendering + Image Transmission	47
Total	189