ABSTRACT

Title of dissertation: HIGH PERFORMANCE AGENT-BASED MODELS WITH REAL-TIME *IN SITU* VISUALIZATION OF INFLAMMATORY AND HEALING RESPONSES IN INJURED VOCAL FOLDS

Nuttiiya Seekhao
Doctor of Philosophy, 2019

Dissertation directed by: Professor Joseph JaJa
Department of Electrical and Computer Engineering
University of Maryland, College Park

Dr. Nicole Yee-Key Li-Jessen
School of Communication Sciences and Disorders
McGill University, Montreal, Québec, Canada

The introduction of clusters of multi-core and many-core processors has played a major role in recent advances in tackling a wide range of new challenging applications and in enabling new frontiers in BigData. However, as the computing power increases, the programming complexity to take optimal advantage of the machine's resources has significantly increased. High-performance computing (HPC) techniques are crucial in realizing the full potential of parallel computing. This research is an interdisciplinary effort focusing on two major directions. The first involves the introduction of HPC techniques to substantially improve the performance of complex biological agent-based models (ABM) simulations, more specifically simulations that are related to the inflammatory and healing responses of vocal folds

at the physiological scale in mammals. The second direction involves improvements and extensions of the existing state-of-the-art vocal fold repair models. These improvements and extensions include comprehensive visualization of large data sets generated by the model and a significant increase in user-simulation interactivity.

We developed a highly-interactive remote simulation and visualization framework for vocal fold (VF) agent-based modeling (ABM). The 3D VF ABM was verified through comparisons with empirical vocal fold data. Representative trends of biomarker predictions in surgically injured vocal folds were observed. The physiologically representative human VF ABM consisted of more than 15 million mobile biological cells. The model maintained and generated 1.7 billion signaling and extracellular matrix (ECM) protein data points in each iteration. The VF ABM employed HPC techniques to optimize its performance by concurrently utilizing the power of multi-core CPU and multiple GPUs. The optimization techniques included the minimization of data transfer between the CPU host and the rendering GPU. These transfer minimization techniques also reduced transfers between peer GPUs in multi-GPU setups. The data transfer minimization techniques were executed with a scheduling scheme that aims to achieve load balancing, maximum overlap of computation and communication, and a high degree of interactivity. This scheduling scheme achieved optimal interactivity by hyper-tasking the available GPUs (GHT). In comparison to the original serial implementation on a popular ABM framework, NetLogo, these schemes have shown substantial performance improvements of 400x and 800x for the 2D and 3D model, respectively. Furthermore, the combination of data footprint and data transfer reduction techniques with GHT achieved high-

interactivity visualization with an average framerate of 42.8 fps. This performance enabled the users to perform real-time data exploration on large simulated outputs and steer the course of their simulation as needed.

HIGH PERFORMANCE AGENT-BASED MODELS WITH REAL-TIME *IN SITU* VISUALIZATION OF INFLAMMATORY AND HEALING RESPONSES IN INJURED VOCAL FOLDS

by

Nuttiiya Seekhao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Joseph F JaJa, Chair/Advisor
Dr. Nicole Yee-Key Li-Jessen, Co-Advisor
Professor Donald Yeung
Professor Gang Qu
Professor Alan Sussman

Dedication

*To my **mother** and **father***
For your unconditional love and care

*To my **life partner***
For your encouragement and support
And for always believing in me, even when it was hard for me to believe in myself

*To my **aunt***
For always being there for me through all my ups and downs

*To my **great aunt***
For sacrificing so much to get me where I am

*To all my **teachers** and **mentors***
For your guidance throughout this long journey

*To all my **puppies***
For the therapeutic stress-relieving petting sessions

*... And to my **beloved grandmother***
For everything you have given me
For always looking after me from up there
***I love you**.*

Acknowledgments

I would first like to thank my advisor, Professor Joseph JaJa for his guidance and patience throughout this long journey. In addition to advising me academically, he has also shared profound ways to look at life and everything in it. The knowledge one could learn from this man is endless. He always put his students' success first and treats us with respect. I consider myself extremely fortunate to have been under his supervision.

I would also like to thank my co-advisor, Dr. Nicole Y. K. Li-Jessen for granting me the opportunity to work with the McGill voice lab. I have learned so many academic and professional skills working with a group of such diverse academic backgrounds. Her guidance and encouragement got me through some of the most difficult times in my graduate career. I am very grateful for the invaluable time she and Professor Luc Mongeau spent reviewing my manuscripts. It was an honor to get to work under such resourceful and knowledgeable people.

My colleagues have given me so much support and deserve a special mention. I would like to thank Caroline Shung, the first person I worked with on the project, for all the effort she put in getting me up to speed when I joined the project. I also would like to thank Kimberly Trickey and Samson Yeun for their hard work improving the project's maintainability. Yu Jin, for his advice on clustering. Lastly, Grace Yu, for her remarkable job translating the literature information to something I can implement, and for putting up with my many requests throughout our collaboration.

I would like to thank those that have mentored me. Dr. Sujal Bista, for

his guidance in developing the visualization component and for his writing advice. Dr. Clifton E. Yu and Dr. Zeynep Dilli, for their extraordinary ability in teaching complex concepts in surprisingly simple ways. And Yimin Zhang, for sharing his wisdom and engaging in countless thought-provoking conversations with me.

I would also like to acknowledge Yun (Yvonna) Li and Alireza Najafi Yazdi for their contributions to the development of the initial and base sequential models. UMIACS staffs, for the hardware support and assistance in VirtualGL configuration.

I would also like to thank the referees for their valuable comments and helpful suggestions for all my publication submissions.

I would like to acknowledge financial support from the National Institute of Deafness and other Communication Disorder of the National Institutes of Health, the Candian Institutes of Health Research, and National Science Foundation for all the projects discussed herein.

I owe my deepest thanks to my family. My mother who has dedicated her life to care for me. My grandmother, for her care and love. Yelly Ruth, for making me a better person and for leaving such wonderfully positive imprints in my life. My life partner and my aunt who always stood by me, even at my lowest point. My great aunt for the all the troubles she had to go through for me. My father for being understanding of everything that I do. Uncle Richard Ruth for always sharing his wisdom and engaging in philosophical conversions with me. My siblings and cousins, Varit Seekhao, Pat Junvisetsak, Delray Promesuwan for always being there for me. Heather Kern, for proofreading some of my work, and for being such an awesome family counselor. My uncle, other relatives and friends in Thailand

who have always provided me with a warm welcome whenever I visit. Words cannot express the gratitude I owe them. I would also like to thank John Taylor and Carol Talkov who are like family members to me. I would like to express my gratitude towards Hui Zhang for his friendship and support. And last, but not least, all the family pups, Richie, Hana, Macy, Lizzie, Happy, Foxie, Teenie, Mokiki, Kirby and Warby, for being my therapists whenever I feel down. You guys are awesome.

It is impossible to remember all who have touched me life, and I apologize to those I've inadvertently left out.

Thank you all so much for coming into my life and enriching it in so many ways. I wouldn't have come this far if it weren't for your love and support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABM	Agent-Based Modeling
Bio-ABM	Biological System Agent-Based Modeling
CPU	Central Processing Units
CUDA	Compute Unified Device Architecture
FFT	Fast Fourier Transform
FPS	Frames per Second
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Units
HADC	Host-Device Activity Aware Data Copy
HPC	High-Performance Computing
LP	Lamina Propria
MPI	Message Passing Interface
OpenACC	Open Accelerators
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
VF	Vocal Fold

Chapter 1: Introduction

In recent decades, the world of computational science has experienced a phenomenal progress. Driven by an unprecedented computing power and the ever-increasing human thirst for knowledge, the complexity of scientific simulations and computations has increased substantially. At the end of Dennard scaling, which assumes constant power density as transistors get smaller [2], a prominent way to keep up with the performance demand has shifted towards multi-core and many-core computing architectures. As opposed to the automatic performance gain through higher clock frequencies achieved before the Dennard scaling ended, the performance gain in the post-Dennard era has been achieved through parallel execution. This change has placed a significant responsibility for performance improvement on the programmers. Well designed parallel programs are essential in leveraging the available resources on parallel architectures. There are libraries and language extensions such as OpenMP [3] and OpenACC [4] available to make parallel programming easier. However, the simplicity through high level abstractions comes at the expense of a decrease in control the programmer has over her program. In many cases, less control is not a problem as long as an acceptable speedup is achieved. Nonetheless, in other cases, the programmer may need the best possible performance to achieve

her goal. In these cases, different aspects of high-performance computing (HPC) designs such as appropriate parallelization, programming models, available resources, scheduling and data distribution need to be considered.

The popularity of Graphics Processing Units (GPUs) has been rising in the scientific computing community. The computing power of massive data-parallelism offered by GPU has been utilized by researchers to enhance the performance of data-intensive applications across different fields [5, 6]. For example, in machine learning and data mining, accelerations through single-GPU and multi-GPU systems have resulted in order of magnitudes of performance improvement over CPU implementations [7]. However, as the data size grows, one limiting factor that interferes with scalability of GPU algorithms is its memory capacity. Fortunately, GPU devices are not stand-alone and they need a CPU host to operate. Hence, the relationship between the host and its accelerators is not mutually exclusive, but rather complementary if utilized appropriately. Though the speed of CPU-GPU interconnect (PICe) is usually the main bottleneck, a carefully designed heterogeneous computing algorithm can mitigate this overhead and offers joint advantages from both architectures through alleviation of individual limitations. For example, communication-computation overlap has often been used to mask the CPU-GPU data transfer latency [8–10] in attempts to achieve optimal throughput.

Computational medicine is one of the areas that has been yielding tremendous benefits from the HPC advancements [11]. High-fidelity models are usually both computationally- and data-intensive. The adoption of HPC in the computational biology community has given rise to many complex models at scales not possible

through traditional sequential algorithms. As computer models are becoming the cost-effective method of choice for developing, testing, validating, and instigating new findings and theories, HPC has become the method of choice in accelerating these models. Despite the number of HPC developments in computational medicine, there is very little work attempting to fully utilize both CPU and GPUs simultaneously. This gap leaves a window of opportunity for designing efficient heterogeneous computing algorithms for large-scale biological physiological system simulations.

This dissertation focuses on tackling the complexity of biological simulations through the development of HPC techniques that take full advantage of the power of heterogeneous CPU-GPU(s) platforms. More specifically, this work targets the modeling of inflammation and repair process of vocal folds (VF) using an agent-based modeling (ABM) approach. The ABM approach was used due to its bottom-up structure that makes cell modeling and incremental knowledge aggregation relatively simple [12]. The inflammation and repair process is multi-scale in nature, involving cellular-level and molecular-level operations and interactions [13]. Biological cells were modeled using autonomous agents. These cells communicate via chemical signaling and perform cellular-level processes such as migration, chemical secretion, and tissue healing in a spatially-discretized environment. The developed HPC techniques were designed in a top-down approach starting from task scheduling, task execution down to data movements. The rest of this chapter briefly discusses the major contributions of this work and outlines the organization of this dissertation.

Contributions This dissertation describes a set of novel HPC algorithms to optimize the execution of complex biological simulations at task-scheduling level,

task-execution level, down to host-device data movement level on heterogeneous CPU-GPU(s) computing platforms.

At the task-scheduling level, three original task-device mapping strategies were developed. First, a simple strategy was designed to enhance hardware resource utilization of 2D VF ABMs by overlapping CPU and GPU numerical simulation tasks. The scheduling strategy was then extended to further overlap visualization tasks, as the visualization tasks became noticeably time consuming in the much larger 3D models. Finally, task-scheduling strategy for large 3D models was restructured to accommodate GPU hyper-tasking (GHT). The ability to hyper-task the GPUs increased the visualization interactivity by allowing the non-rendering GPUs to participate in the visualization process of their local data and minimize the amount of off-device data transfers.

At the task-execution level, OpenMP and custom thread-safe concurrent data structure were used to speed up the cellular-level processes. Chemical diffusion computations were then optimized using convolution-based diffusion to advance multiple iterations of chemical diffusion in a single step of computation. Furthermore, the performance of the visualization component was enhanced through data access reduction techniques including adaptive sampling and activity-aware host-device data transfers.

The coupling of task-scheduling and task-execution level HPC techniques for 2D VF ABM resulted a significant speedup of 35x and 7x over sequential and OpenMP versions, respectively. The 3D VF ABM achieved 2.4x the throughput of 2D VF ABM. More specifically, the 3D VF ABM was capable of simulating 17

million biological cells in under 7 seconds per iteration, and visualizing time-varying volumetric output of 1.7 billion protein data points with an average framerate of 42.8 fps. This performance improvement results in real-time interactivity that allows users to explore large simulated data sets on a remote server without any lag. Our simulation platform also offers computational steering capability as the visualization is performed *in situ*. The ability to simulate, explore data and perform computational steering in real-time can assist modelers and researchers significantly in testing and validating their hypotheses and new scientific discoveries.

Dissertation Organization This dissertation consists of 7 chapters. The next chapter (Chapter 2) provides a background and overview of the fields surrounding this work including HPC, systems biology simulation and 3D visualization of biological systems. Chapter 3 - 5 discuss the details of HPC techniques used in this work, which were adapted from our previous publications [14–17]. More specifically, Chapter 3 discusses the techniques used to achieve optimal concurrency at the scheduling level. The chapter begins with the explanation of the task categorization method which became the basis of heterogeneous platform task mapping strategies used in this work. The chapter then expands on three different task-device mapping strategies. Chapter 4 discusses the numerical simulation optimizations for both cellular- and molecular-level processes. First, parallelization of cellular level processes using OpenMP on CPU is discussed, followed by a discussion on rationale, implementation and optimization of convolution-based diffusion using fast Fourier transform (FFT).

Chapter 5 describes the optimization techniques developed to enhance the

visualization performance and interactivity. First, the importance of *in situ* visualization and the protocol used in the work is discussed. The chapter then establishes data access pattern types, which were essential in further development of host-device data transfer minimization using an activity-aware redundancy reduction scheme. Finally, a heuristic for choosing best parameter (block size) for this Host-Device Activity Aware Data Copy (HADC) is proposed.

Chapter 6 starts with model configuration details to give an overview of the simulation scale. The chapter then details the performance gain achieved in 2D, 3D and high-interactivity 3D models, through different HPC techniques developed. Since different forms of sampling were used to reduce GPU memory requirements, the trade-offs between accuracy and performance were studied. The performance gain using parameter recommendation heuristic for the HADC technique was measured and compared against the best parameter. Lastly, the details regarding model verification are discussed.

Chapter 7 provides conclusion and discusses future directions of this work.

Thesis Statement The product of this dissertation is a set of heterogeneous computing algorithms applicable to discrete-time multi-scale models of complex cellular-level systems. Designed to run efficiently on CPU-GPU(s) heterogeneous computing platforms, these techniques can be applied to cellular-level models to significantly improve the performance of both numerical simulation and visualization through efficient task scheduling, task execution and host-device data management. Demonstrated through the case study of vocal fold repair model, the HPC

methods developed in this dissertation enable real-time simulation and *in situ* visualization of large-scale complex biological systems, offering computational steering and highly interactive time-varying volumetric data exploration capability at the scale not currently present in other related work. The computational steering and real-time data exploration capabilities allow biomedical researchers to implement, test and refine their cellular-level models. This ability to perform iterative model refinement can continuously improve the accuracy and ultimately lead to clinical quality computational models.

Chapter 2: Background

To provide an overview of the fields surrounding this dissertation, this chapter begins with a brief discussion of the main HPC hardware and software paradigms. The chapter then provides the details of the simulation approach used, followed by the background on the specific case study of vocal fold inflammation. Existing work in the fields of biological system modeling, HPC ABM in bio-applications and visualization in Bio-ABMs will be discussed briefly. Finally, the chapter ends with a description of the hardware and software environment employed in this work.

2.1 Parallel Computing Architectures

In this section, some of the relevant trends in computing hardware will be discussed. First, a brief background on evolution of popular processors such as microprocessor (CPUs) and coprocessor/accelerator (GPUs) will be provided. Next, we discuss the trends of both computing clusters and personal computers. The main focus of this discussion will be on accelerators as x86 seems to have been the dominant microprocessor architecture for many years [18], thus no substantial changes have occurred in terms of microprocessors. Specifically, we will use GPU as the representative coprocessor architecture because it is the most common type of

accelerators [18].

2.1.1 Multi-Core Central Processing Units (CPUs)

Driven by a performance hungry market, there is always a demand for faster processors 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 [19]. And continuous performance improvement of a processor has been relying on the increase in density of integrated circuits (ICs) on a chip for decades [20, 21]. However, according to Pollack's rule, performance increase by microarchitecture alone is roughly proportional to square root of increase in complexity [22], 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 [23]. And with higher integrated density, power dissipation becomes the bottleneck of the architecture [22, 23]. 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 [24]. Multi-core architecture allows for a scalable processor design and offers a way to achieve better performance without infringing on the power dissipation requirements [22–24].

Today, a server-grade CPU chip can consist of up to 56 CPU cores [25]. A powerful compute node may consist of multiple CPU sockets resulting in more cores.

For more computing power, multiple compute nodes can work together in a cluster and communicate among themselves via high-speed interconnection networks.

2.1.2 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 was then introduced in 2007 to enable GPGPU programming in C with C-like extensions (for more details please refer to Section 2.6). Since its introduction, more than 100 million computers with CUDA-capable GPUs have been shipped to end users [26].

GPUs consist of a number of Streaming Multiprocessors (SMs), each of which contains a number of processor cores (e.g. stream processors or CUDA cores). Each SM consists of tens of thousands of registers used by GPU threads during kernel executions. The SMs also contain different levels of caches including L1 caches, texture caches, constant caches and shared memory [27]. A GPU typically contain thousands of cores making GPUs capable of launching thousands of threads simul-

taneously. All the SMs have access to the high bandwidth Device memory (peak bandwidth 240 GB/s). The best bandwidth is achieved through data coalescing. Coalesced memory access refers to a scenario where number of memory transactions is minimized when consecutive threads access consecutive memory locations.

Graphics processing hardware has existed since the 1990's [28], long before the term *GPU* was even devised by NVIDIA in 1999 with the launch of its GeForce 256, a graphics accelerator hardware with 23 million transistors [29]. The dedicated graphic cards offered new graphics features and huge performance improvements, resulting in tremendous interests from the gaming community [30]. Since then, more and more personal computers picked up on the host-accelerator architectures for their immense graphics capability. The prices of these units also play a role in their popularity as decent GPUs are becoming more affordable [31]. Today, one of the most popular graphics card, NVIDIA GeForce GTX 1060 [32], with 4.4 billion transistors on 1280 cores, costs less than $300 [33]. As the demand for graphics complexity increases with the performance per dollar, dedicated GPU hardware is present in most personal machines used by gamers, film editors and anyone serious about utilizing computer graphics capability and performance.

2.1.3 Heterogeneous Computing

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 [34]. Heterogeneous plat-

forms of various architectures and scales have become increasingly popular in both HPC and personal computing platforms. For example the larger scale platforms are based on large clusters of different types of multicore CPUs and many-core accelerators such as Graphics Processing Units (GPUs) [35–37], Field-Programmable Gate Arrays (FPGAs) [35, 38], while FPGA and Digital Signal Processors (DSPs) are often seen in power critical systems [39, 40]. 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. However, this requires a careful scheduling and orchestration of the operations using the available resources. For example, to mask the CPU-GPU data transfer latency in out-of-core GPU memory management for MapReduce-based large-scale graph processing, researchers have proposed a technique to overlap computations and data-transfers [8]. In addition to computation-communication overlap, another heterogeneous computing larger-scale graph processing framework, HyGraph [10], achieved performance improvements over its cpu-only and gpu-only counterpart through dynamic scheduling. A major part of this dissertation focuses on optimal collaboration of a multi-core CPU with one or more GPUs on a single compute node. This increases the applicability of

the contributions of this work to other applications, as CPUs-GPUs platforms are pervasive today.

2.1.4 Computing Clusters

A computing cluster is a network of computers that work together to form a single high-performance distributed system. These computers are usually connected via fast local connections such as Infiniband™ and Gigabit Ethernet interconnects. As microprocessors get cheaper and network interconnects become faster, computing clusters offer a solution to computationally intensive applications by scaling out commodity machines. There is a wide spectrum of computing clusters ranging from small business clusters to large supercomputers.

Accelerators are becoming more and more popular in squeezing the teraflops-per-second into supercomputing systems. In 2010, there were only 10 GPU-based systems in the top 500 list. Despite the fluctuation, the overall trend has been favorable towards accelerators from different architecture families. For example, in 2012, NVIDIA GPUs alone account for 50 systems in the list, and that number has gone up to 64, 85 and 128 in 2015, 2017 and 2018 [41], respectively. As of November 2017, out of the top 50 HPC applications, more than two-third offer support for GPU-based acceleration with more under development to offer this feature [42].

2.2 Parallel Programming Models

Parallel programming models are built on top of parallel hardware and physical memory architectures. Theoretically, any type of parallel programming model can be implemented irrespective to the underlying hardware architecture, though efficiency may decrease if the programming model and underlying hardware are not somewhat compatible. As in parallel computing architectures, there are also multiple ways to classify parallel programming models. This section will focus on the two widely used parallel programming communication models (shared and distributed memory), data-parallel model (SPMD), hybrid models and the rapidly emerging model, MapReduce.

Shared Memory Parallel Programming (SMP) — Shared Address Space Model. When memory is global and all threads have the same view of memory, we are in a *shared address space*. This programming model makes it easy for thread to communicate and modify the states of the global memory. However, if synchronization is not implemented correctly, the program will not execute correctly. Incorrect synchronization can cause problems such as data-races and dead-locks. Examples of shared memory APIs include POSIX-thread (pthread) [43] and Open Multiprocessing (OpenMP) [3].

Distributed Memory Parallel Programming — Message Passing Model. In distributed memory model, each process has their own memory space which is not directly accessible or visible to other processes. Thus, message passing communication model is used in this type of memory model for processes to communicate

with each other and requesting state changes in others' memory spaces. This type of memory is mostly present in computing clusters, as multiple commodity hardware are connected to each other via interconnects. The Message Passing Interface (MPI) forum was formed in 1992 and released the first part of MPI specification in 1994, with the following parts in 1996 and 2012, respectively [44–47]. MPI implementations exist for most, if not all, modern parallel computing systems, however, some of the implementations may not support all functionality specified by MPI-1, MPI-2 and MPI-3.

Data Parallel Programming Model (SPMD) — In this model, the program performs the same operations on different parts of a data set. Historically, data parallel programming model were mostly used to perform the same instructions to each array element such as computations seen in vector processors [48, 49]. For example, `add(A, B, n)` computes, in parallel, an addition of each of the `n` elements of vector (array) `A` to the corresponding element in vector `B`. In modern days, data parallelism is mostly seen in the Single Program Multiple Data programming model. An example includes GPU computing, which uses the same program to process multiple data elements concurrently [50]. More examples include partitioned global address space (PGAS) languages such as Unified Parallel C (UPC), X10 and Chapel [51]. PGAS assumes a global memory view that is partitioned [52]. Each partition of the global memory belongs to a program thread. In PGAS, all threads execute the same program which operates on each thread's own memory partition.

Hybrid Memory Parallel Programming — This model combines more than one of the previously mentioned parallel programming models together. For

example, on a cluster of multi-core compute nodes, distributed and shared memory programming can be combined by allowing threads to view and modify their local memory with OpenMP and communicate with processes from other compute nodes via MPI. Another example would be using pthread to parallelize the execution of certain compute tasks in a shared-memory manner on a multi-core CPU and offload computationally intensive compute tasks to a GPU. The hybrid model allows the programmer to strive for "the best of both worlds" model of parallel programming, given their specific application needs.

MapReduce — Developed and originally used by Google, MapReduce has continuously been gaining traction from the big data community since its first publication in 2008 [53]. Since then, MapReduce has been featured as the parallel programming model of choice to enhance the performance of many big data applications in fields including machine learning, data mining and bioinformatics [54–57].

MapReduce takes in a set of $input < key, value >$ pairs, and produces a set of $output < key, value >$ pairs. The programmer expresses their computations in MapReduce by writing two functions: map and reduce. The map function written by the programmer produces a set of $intermediate < key, value >$ pairs. The underlying framework *shuffle*s these intermediate pairs, group them by their intermediate keys, and pass them to the reducer. To process large amounts of data in a distributed fashion, the underlying distributed file system (DFS) was designed to handle machine failure and offer fault tolerance through redundant execution and data replications. In 2011, Apache Software Foundation released its first version of fault tolerance DFS, Apache Hadoop. As opposed to the Google File System (GFS),

Hadoop is open-source, thus allowing more access to the MapReduce programming model.

Despite its applicability to certain big data applications, MapReduce is not suitable for iterative algorithms as all inputs and outputs of the compute stages require disk access. This has put limitations on the performance enhancement of multiple applications requiring iterative data accesses including training algorithms for machine learning models. This limitation of MapReduce has motivated the development of Apache Spark [58]. Since its released in 2014, Spark has been gaining significant interest from the big data and machine learning community as it mitigates the limitations of MapReduce by allowing in-memory processing, thus alleviating I/O overhead [59].

2.3 Agent-Based Modeling (ABM)

Agent-based modeling is a widely adopted approach to quantitatively simulate dynamical systems [60]. The popularity of ABMs can be observed in the variety of ABM frameworks developed in the past decade (for reviews, please see [60–63]). This modeling approach abstracts the system of interest by using a set of autonomous objects, or ABM *agent*s, to execute the decision-making behavior of individual components of the system. These ABM *agent*s interact among themselves, as well as with their environment according to a number of predefined stochastic and/or deterministic rules [60, 62]. In contrast to equation-based approaches, ABMs are decentralized. That is, the system's behavior is determined by the collective behavior

of each individual *agent* in the system. Although a universal definition of ABMs remains debatable [60], fundamental components of ABM typically include: agent set, agent relationship set, and agents' environment [12].

Firstly, a set of agents includes the agents themselves, their attributes and their behavioral rules. Agents' behavioral rules govern their decisions and actions. In ABM, *agent*s can represent a wide spectrum of individual entities such as consumers, markets and geographic regions in economic models [64–69], animals in ecosystems [70–73] and biological cells and proteins in systems biology models [1, 14, 74–83]. Secondly, the set of "agent relationships and methods of interactions" [12] defines the criteria of a group of entities each agent is bound to interact with, and how these interactions are carried out. For instance, some ABMs may allow agents to interact only *directly* with other agents, some may allow only *indirect* interactions while some may allow both [84]. A *direct* interaction represents an immediate impact one agent leaves on another. Particle collision is an example of a direct interaction, where colliding particle agents affect the states of each other directly. On the other hand, *indirect* interactions have been used to mimic the lingering effects of transmitted signals [85–88]. An example of indirect agent interaction includes chemical secretion as a form of inter-cellular communication. This chemical secretion example is classified as indirect because the agents alter the states of the environment to communicate, rather than altering the states of the recipient agents directly. Lastly, the agents' environment houses the autonomous agents. This space can be discrete lattice-based [89], continuous lattice-free [90] or hybrid [91]. The environment may maintain local attributes depending on the

application and underlying implementation [92].

Our collaborators' first published ABM [74] was programmed on the platform of Netlogo and thus most of the terminology used herein was adopted from the dictionary of NetLogo [93]. In both 2D and 3D implementations described in this work, the simulation environment, also known as the ABM *world*, represents human tissue. The environment is spatially discretized into rectangular volumes called *patch*es. Each mobile *agent* represents an inflammatory cell that can move from one patch to an adjacent patch and make decisions to perform certain actions at discrete time steps. *Agent*s make decisions based on the state of the *patch*es, which allow them to alter their environment to interact indirectly with other *agent*s. Chemokines and extracellular matrix (ECM) proteins are associated with the states of the patches.

2.4 Vocal Folds Injury Model

2.4.1 Problem Background

Voice problems were estimated to affect one in 13 adults in the United States annually [94]. In one study, nearly one third of the sampled population has experienced voice disorder symptoms at some point in their lifetime [95]. In particular, voice disorders constitute a major occupational hazard in many professions such as salespeople, teachers, performing artists, attorneys, and sport coaches, due to the intensive vocal demand of the job [96–100]. The estimated lifetime prevalence of voice disorders is as much as 80% in occupational voice users [101–103]. Human vocal folds are under continuous biomechanical stress during voice production. Exces-

sive phonatory stress can induce a cell-mediated inflammatory response and structural tissue damage, leading to a pathological condition [104–109]. Patients with phonotraumatic lesions are usually prescribed behavioral voice therapy [105, 110] or surgical excision of the lesion in combination with various adjunctive treatments [111–117]. Unfortunately, the healing outcome of voice treatments often depend on the lesion, the treatment dose, and the patient's vocal needs [74, 118–121]. The success rate of voice treatment varies extensively between 30% and 100% [122–127] , making the treatment planning process difficult for voice therapists and surgeons. The unpredictable treatment outcome is axiomatic and takes a huge toll on a person's career, a clinician's decision-making process and society's healthcare costs. A predictive tool that can estimate voice treatment success would spare patients from unnecessary and costly treatments and potentially harmful side effects.

A series of agent-based models (ABM) have been developed to numerically simulate the essential biology underlying vocal injury and repair that may help clinicians to better tailor treatments for patients with voice disorders [1, 14, 74, 77, 128–130]. The flow diagram (modified from [74]) of the interactions between all the components in the model is shown in Fig 2.1. We first developed 2D vocal fold ABM [14] to simulate inflammation and repair of a single vocal fold tissue slab. The 2D vocal fold ABM was then upgraded to a much larger 3D model to resemble the physiological dimension of human vocal folds. Techniques including diffusion kernel reduction and data copy minimization were used to boost the performance of both the computation and visualization executions. These techniques were coupled with a scheduling scheme that completely masks the execution time of the computation-

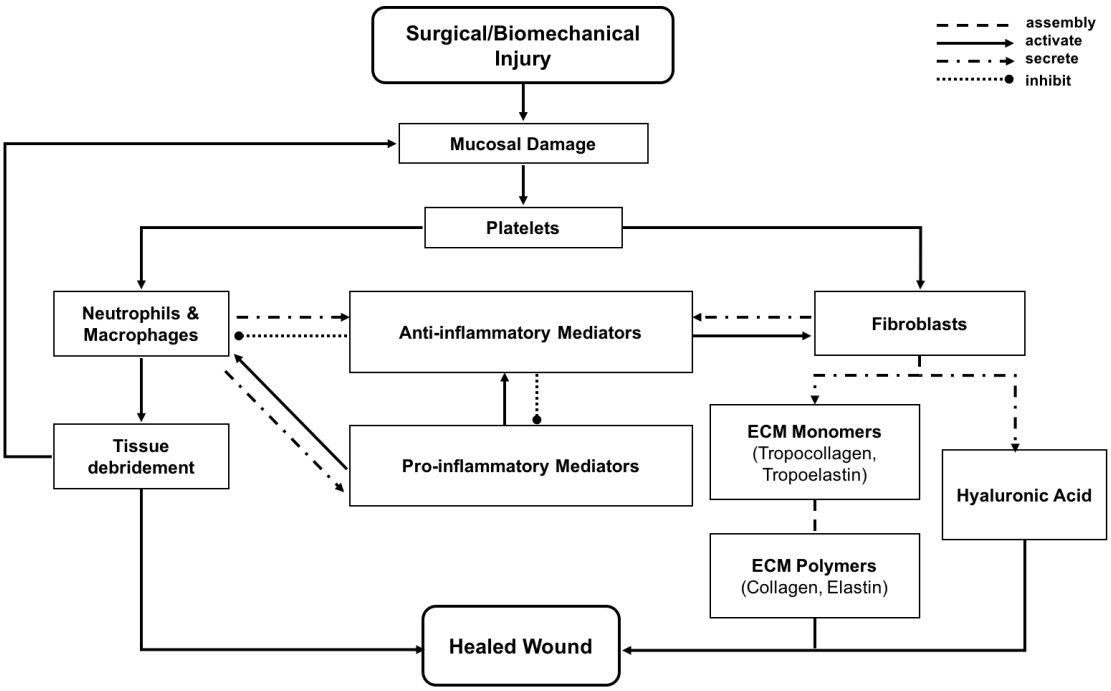


Figure 2.1: Flowchart of vocal fold inflammation and healing events in ABM. This diagram is taken from [74].

ally demanding diffusion and visualization tasks on heterogeneous compute nodes consisting of multi-core CPU and multiple GPUs. This low-cost, high-resolution and high-performance computing ABM platform with real-time visualization capability is original in disease modeling and necessary to make complex disease models executable and practical in clinical settings.

A vocal fold (VF) ABM simulating inflammation and repair was developed and partially verified against empirical vocal fold trauma data [1, 74, 77]. Inflammatory cells were implemented as ABM *agents*, while chemokines and extra-cellular matrix (ECM) proteins were implemented as states of the ABM *patches*. The aggregation of these ABM components yields the state of the vocal fold (ABM *world*) at each

given point in simulated time. Table 2.1 summarizes the roles that each type of cell agents plays in the healing process. At the time of acute injury, the traumatized mucosal tissue within the damaged area triggers platelet degranulation [1, 77]. Different chemokines get secreted resulting in vasodilation stimulation and attraction of inflammatory cells, namely, neutrophils and macrophages to the wound site. Activated neutrophils and macrophages at the wound area further secrete chemokines to attract fibroblasts and clean up cell debris. To repair the wound, activated fibroblasts proliferate and deposit extracellular matrix (ECM) proteins such as collagen, elastin, and hyaluronan. These ECM proteins then form a scaffold for supporting fibroblasts in wound contraction and other cells migration and wound repair activities [131].

Agent	**Actions**
Platelets	Secrete TGF-β1, MMP8 and IL-1β to attract other cells.
Neutrophils	Secrete TNF-α and MMP8 to attract other Neutrophils and Macrophages.
Macrophages	Secrete TNF-α, TGF-β1, FGF, IL-1β, IL-6, IL-8, IL-10 to attract Neutrophils, other Macrophages and Fibroblasts. Clean up cell debris.
Fibroblasts	Secrete TNF-α, TGF-β1, 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 2.1: Summary of agent rules

2.4.2 Computational Challenges and the Need for HPC in Bio-Simulation

A major challenge of cellular-level simulations for biological systems is dealing with large data sets. For accuracy, a biologically representative *world* size and

appropriate high-resolution spatial discretization are crucial. In most cases, the finely discretized simulation space results in the need to maintain billions of dynamic data points. Thus, high-performance computing (HPC) techniques, language and hardware resources are needed.

Another significant challenge in systems biology modeling lies in the multi-scale nature of the model [79, 132–137]. To ensure optimal performance, it is important for differences in spatiotemporal scales between cellular and chemical interactions to be handled in a cost-effective manner. Cellular movements occur at a rate of micrometers per hour ($\mu m/h$), while cytokine diffusion in tissue occurs at a rate of micrometers per second ($\mu m/s$). A naive approach would be to iteratively simulate the model at the smallest temporal scale required. However, this approach would result in a prohibitive increase in the computational cost. A possible solution is to use coarse-graining techniques to lower the computational intensity [138]. The concept of coarse-graining in ABM refers to the simulation of *super-agents* whose rules represent aggregated behaviors of smaller units [139–141]. Our ABM frameworks use a mechanism that captures the behavior of multiple iterations of the finer-scale processes, i.e. chemical diffusion, over a coarse time window using convolution [14, 16]. This intensive computation is then offloaded to a single GPU while the CPU cores focus on coarse-grain cellular processes.

High-fidelity models produce billions of output data points per iteration. An effective visualization is the fundamental component to analyzing these outputs by transforming the enormous set of numbers into an intelligible form. It is also as essential that the visualization is set up with a protocol that results in the most

optimal performance. The convention used to be that visualization is performed on pre-simulated/pre-processed data that is stored on disk; this method is known as *post hoc* visualization. However, large simulation data sets have prompted work on co-processing also known as *in situ* [142] visualization in order to move the program to the data as outputs are becoming too heavy and expensive to move around. We have addressed this issue in [14–16], which will be described in Chapter 5.

2.5 Related Work

2.5.1 Biological System Modeling

Computer simulations have become central to personalized medicine [143–146]. This approach involves the creation of computational models to estimate treatment outcome and identify the best possible treatment for a given patient. Simulation modeling involves the integration of the best available knowledge into a computer platform to represent the real-world problem. The process involves an abstraction of causal relationships between patient variables and health outcomes followed by a rigorous and iterative protocol of model calibration and validation [147–149]. The property that sets numerical simulation models apart from standard statistical models is the observability of the evolution of patient behaviors and health conditions in the computer model as time passes during simulation. Such an approach provides a computational tool for clinicians to evaluate the impact of intervention or other modifiable variables on health outcomes in advance or along any point during the intervention.

Computer models have been developed for complex health conditions, including sepsis [150–152], traumatic brain injury [153], acute liver failure [154], diabetes [155, 156], obesity [157, 158], and cardiovascular disease [159–161]. In our case, a series of ABMs have been developed to numerically simulate the essential biology underlying vocal injury and repair with the goal of helping clinicians to better tailor treatments for patients with voice disorders [1, 14, 74, 77, 128, 130].

2.5.2 HPC ABM for Biological Applications

High-fidelity ABMs in biology (Bio-ABM) often involve large amounts of data. Multiple high-performance computing (HPC) ABM tools have been developed to address the challenges in processing these large data sets. For example, FLAME [162, 163] is an implementation of an ABM framework for parallel architectures based on stream X-machines. FLAME has been used to speed up the simulation of ecological systems in various fields including systems biology [76]. FLAME was further extended with a GPU support [164, 165]. SugarScape on steroid [166] is another example of ABM acceleration on GPU platforms. These tools have demonstrated their applicability to biological system simulations such as tissue wound and disease modeling [76, 167, 168]. In the realm of distributed computing, Repast HPC [169] was developed as an MPI extension to its predecessors, Rapast and Repast Symphony [170, 171]. Repast HPC was adopted to accelerate the simulation of bone tissue growth [172].

Due to the popularity of ABMs in biological applications, some HPC ABM

tools have also been developed specifically for biological applications. An example includes AgentCell, a Repast-based framework for single-cells and bacterial population [173]. The AgentCell framework provides support for running multiple non-interacting single-cell instances concurrently on massively parallel computers. Chaste [174] is an example of bio-ABM framework utilizing HPC platforms via various existing high-performance libraries like PETSs and (par)METIS for parallel linear algebra and mesh distribution. More examples include HPC ABM frameworks for multi-core CPUs such as CompuCell3D [175,176], CellSys [177], and Morpheus [178]. In these frameworks, parallelization was performed with OpenMP to speed up the performance on single-node multi-core CPUs. In addition, other techniques have been proposed to accelerate specific biological models on multi-core CPUs or GPUs [179–182]. The aforementioned HPC ABM techniques either target CPUs or GPUs. The inability to exploit both CPUs and GPUs simultaneously results in sub-optimal resource utilization. Our previous work [14, 16] addressed this issue by assigning appropriate computational tasks to the CPU while it waits for the GPUs to complete their computations. However, these previous works only focused on the simulation computation. This leaves an opportunity in visualization optimization an open problem.

2.5.3 3D Visualization in Bio-ABMs

The challenge in developing HPC bio-ABM has been relatively well-explored. However, the same does not apply to the visualization aspect of ABM frameworks.

Thus, many ABM users resort to a more general data visualization tool such as Paraview [183] and VisIt [184].

Due to the scales of the data size being generated, the conventional method of *post hoc* visualization has become increasingly infeasible. The *post hoc* method refers to a visualization process where the visualization is performed after the numerical simulation has completed. The output data are written to disk during the simulation and retrieved later for visualization. This visualization method puts an enormous load on the disk and network. For this reason, another type of visualization, *In situ* visualization, has gained the interests of researchers [185]. *In situ* visualization allows the outputs to be analyzed on the same machine that produced them. The ability to perform on-site data analysis reduces the amount of data movements between the server and remote users. This property makes *in situ* visualization an ideal way to visualize simulations that produce large data sets such as our case. Paraview Catalyst [186, 187] and work reported in [188] are examples of libraries developed to enable *in situ* processing of simulation output on popular existing visualization frameworks such as Paraview [183] and VisIt [184]. A bitmap-based and a quadtree-based ABM approach [81,189] were proposed respectively to analyze the numerical output *in situ* and to reduce non-essential simulation data.

Although atomic-level visualization engines are not visualization frameworks specific to ABM, some of their optimization techniques can potentially be applied to a cellular-level ABM visualization framework. For example, MegaMol [190] is an open-source cross-platform visualization prototyping system. Through low-level GPU optimizations, a case study shows MegaMol's capability in visualizing 100 mil-

lion atoms at the 10 frames per second (fps). CellView [191] is another example of molecular-level visualization framework designed to process large amounts of data. With the assumption that the framework will be used for large bio-molecular landscapes, CellView employed level-of-details (LOD) and culling technique to achieve an impressive frame rate of 60 fps on a 15 billion-particle data set. However, CellView does not support *in situ* visualization. Thus, the effective frame rate may be lower if the framework is coupled with a simulation engine.

2.6 Hardware and Software Environment

The computation component of all versions of our 2D VF ABM were tested and benchmarked on a compute node with 16-core Intel(R) Xeon(R) E5-2690 CPU and NVIDIA Tesla K20c GPU. The complete simulation suite consisting of both computation and visualization components were tested and benchmarked on a compute node consisted of a 16-core Intel(R) Xeon(R) CPU E5-2630 and an NVIDIA Tesla K80 GPU with rendering enabled. All current versions of our 3D VF ABM has been tested and benchmarked on a compute node with 44-core Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz host and two attached accelerators, NVIDIA Tesla M40. The host has 128 GB of main memory. Each Tesla M40 GPU consists of 3072 cores per device with 24 GB of global memory. Table 2.2 summarizes the specifications of each GPU used.

To ensure fast and efficient simulation, a lightweight programming language, C++, is used to implement the program. To utilize the multiple CPU cores available,

GPU	Tesla K20c	Tesla K80	Tesla M40
SMs (per Device)	13	13	24
CUDA Cores per SM	192	192	128
Registers per SM	64k	64k	64k
L2 Cache Size	1.25 MB	1.50 MB	3.0 MB
Global Memory (per Device)	4.7 GB	11.25 GB	24 GB
Max Clock Rate	0.71 GHz	0.82 GHz	1.11 GHz
Memory Clock Rate	2.6 GHz	2.5 GHz	3.0 GHz
Memory Bandwidth	208 GB/s	240 GB/s	288 GB/s
Compute Capability	3.5	3.7	5.2

Table 2.2: Summary of NVIDIA Tesla K20c, K80 and M40 specifications

Open Multi-Processing (OpenMP) was used to parallelize tasks to be executed on the CPU cores. OpenMP is a highly portable Application Programming Interface (API) supporting multi-threading on shared-memory platforms via a set of platform-independent compiler directives [3]. In addition, OpenMP was also used to allocate separate threads to communicate and launch tasks on the GPUs for heterogeneous hardware concurrency. Computationally demanding tasks to be executed on the GPUs are managed using NVIDIA Compute Unified Device Architecture (CUDA) [192] model. CUDA is a parallel computing platform and programming model, which allows general purpose multi-threaded programming of GPUs via C-like language extension keywords. In the CUDA language, a GPU is presumed to be attached to the host (CPU) which controls data movement to/from the GPU. The CPU host is also responsible for launching CUDA kernels, which are functions to be executed by all threads launched on the GPU. Open Graphics Library (OpenGL), an open standard, cross-language API for 2D and 3D rendering, has been used to implement the simulation visualization. OpenGL is widely used over a broad range of graphics applications due to its portability and speed. The pre-processing of data to predict

best parameters for the host-device data minimization algorithm was developed in Python 3.7.

Chapter 3: CPU-GPU Task Scheduling

Hardware resource utilization is an important determining factor for software performance. This chapter discusses the techniques we developed to achieve optimal concurrency of resource utilization at the scheduling level. The chapter begins with the explanation of the task categorization method which became the basis of heterogeneous platform task mapping strategies used in this work. The chapter then expands on task-device mapping strategies designed for 2D, 3D and highly interactive 3D cellular-level ABMs.

3.1 Task Categorization

The compute tasks are divided into two categories; coarse-grain and fine-grain. The coarse-grain tasks, such as inflammatory cell functions and ECM protein functions, are best handled by CPUs as they entail more complex computations that deal with relatively small amounts of data. In contrast, the GPUs are good at the fine-grain tasks such as the diffusion of chemo-taxins, due to the exceptional ability of GPUs to process simple tasks that involve large amounts of data effectively.

3.2 2D Vocal Folds ABM

In our previous publication [14], we proposed a mechanism to hide the diffusion computation time by utilizing our GPU and p CPU cores concurrently. The details of CPU threads' responsibilities are as follows:

i) 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 (discussed later in Section 4.2) 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 3.1.

3.3 3D Vocal Folds ABM

The 3D VF-ABM [16] consisted of an environment with 154 million patches. Each patch stored information of ECM proteins and chemical data. In addition, around 17 million mobile agents, representing the inflammatory cells, resided in this ABM world. The model simulated the dynamic biological processes pertinent

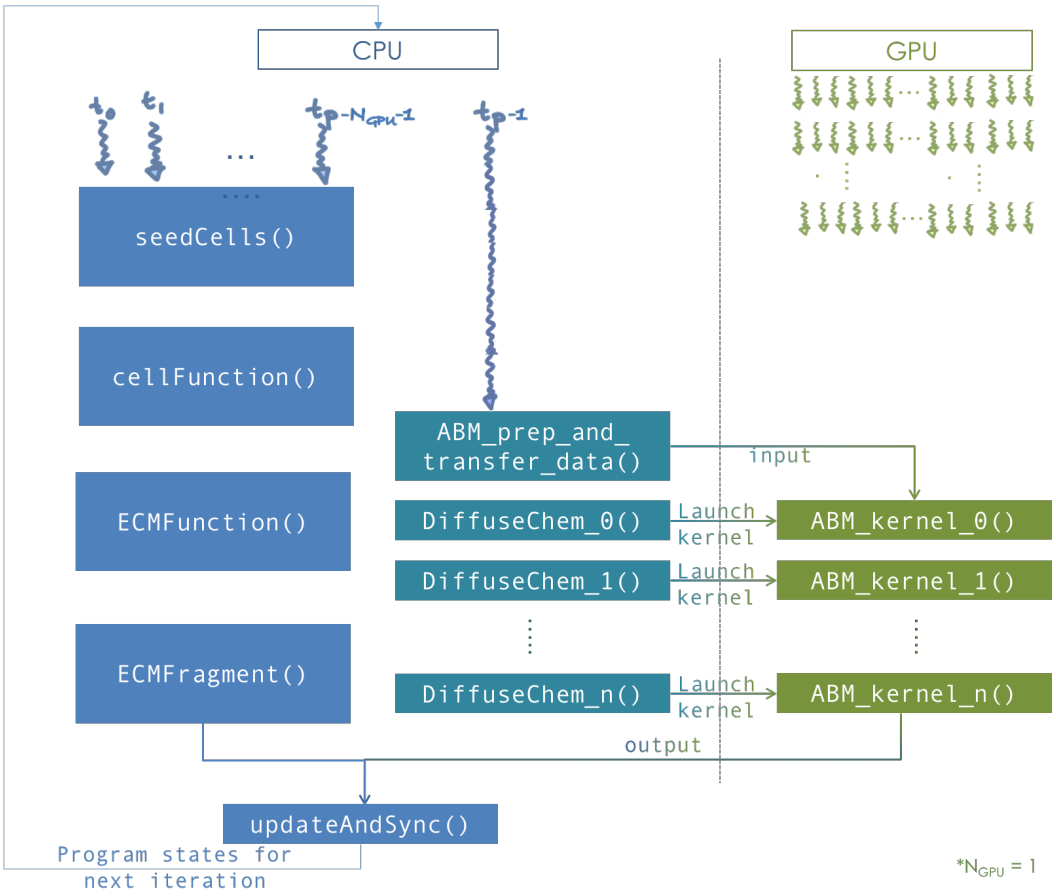


Figure 3.1: Diagram demonstrating CPU-GPU task scheduling scheme for 2D multi-scale ABM

to vocal fold inflammation and repair at 30 minute time intervals. At each model iteration, the operations corresponding to ECM functions, chemical diffusion, and cell (agent) functions were executed, followed by the update of the ABM world. Given the computational complexity and the amount of data involved, each iteration required a careful mapping and scheduling of these operations on the available hardware resources. In addition, the visualization provided essential spatial information of ECM proteins, chemicals, and inflammatory cells during the simulation. The overall goal was to numerically simulate and graphically visualize the 3D VF-ABM as fast as possible for each iteration.

To achieve optimal resource utilization, it is important to address the challenges of load balancing, minimizing data movements between the CPU and GPU, and coordinating the tasks on various devices. As we moved from 2D [14] to 3D [16], the computational complexity of the simulation and the amount of data involved increased substantially. Furthermore, the execution time of the visualization component, which was negligible in the 2D simulation, became significant. Therefore the issues of task assignment, load balancing, and device coordination need to be revisited and addressed properly.

Fig 3.2 illustrates the workflow of the 3D ABM simulation during each iteration. Specifically, it describes the task allocation on a platform consisting of a single multicore CPU with N_{GPU} GPUs attached to it. For our specific setup consisting of 2 GPUs, the simulation started on the CPU host, and then split into 3 paths: coarse-grain, fine-grain/visualization, and fine-grain. Each of the paths was run on separate hardware resources. The first path spawned multiple CPU threads

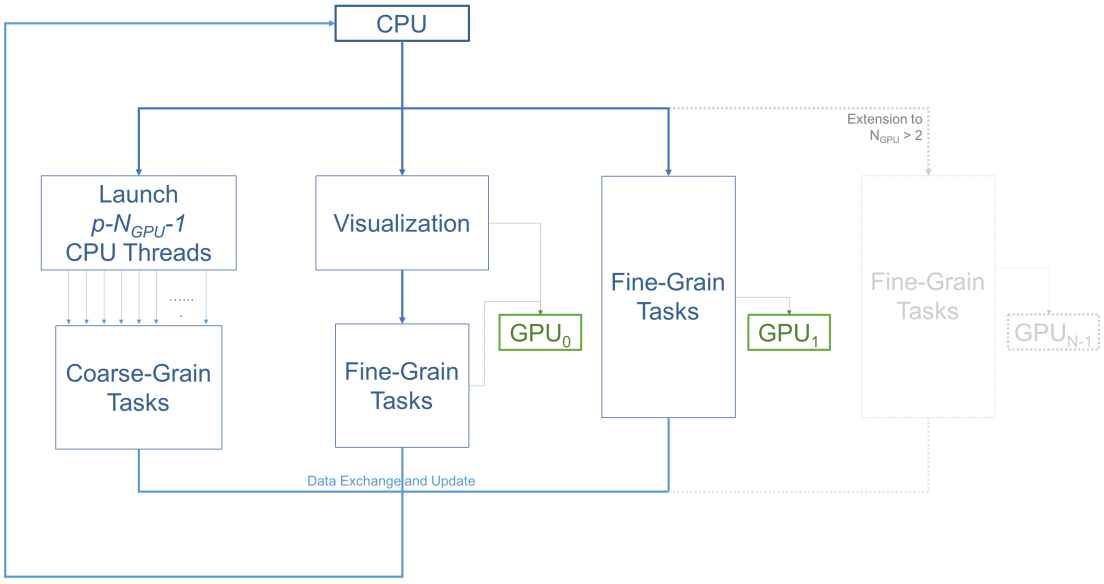


Figure 3.2: Diagram demonstrating CPU-GPU task scheduling scheme for 3D multi-scale VF ABM

to execute coarse-grain tasks on CPU cores. The second path was responsible for visualization and some of the fine-grain tasks that execute on a single GPU resource. The remaining fine grain tasks executed on the rest of the GPUs. All paths met at the end to exchange and update the ABM world.

The overlap of visualization and computational components required a careful device coordination as these components now share computing resources. Algorithm 1 describes, at a high-level, how to map tasks and perform host-devices synchronization. Each GPU task, computational or visualization, has its own CPU thread for data management and communication with the GPUs. Nested CPU threads were launched at three levels. At the first level, the driver started the execution by initializing the simulation and launching two threads, one for visualization and the other for computation. The visualization rendered the current state

of the ABM world using an available GPU, and then broadcasted the completion of the rendering task. Concurrently with the visualization execution, the computation started by launching two more threads at the second level. Both threads at this level further launched multiple threads at the third level, depending on the number of cores available. More specifically, the first thread at level 2 was responsible for executing CPU tasks, which launched parallel threads for coarse-grain task parallelization i.e. level 3. The second thread at level 2 spawned N_{GPU} level-2 threads to launch fine-grain computation tasks on available GPUs. Note that if the visualization was not yet completed, one of the GPUs would not be available and the fine-grain tasks would have to wait (Algorithm 2). If a fine-grain task had grabbed the same GPU used for visualization, it would have to broadcast its completion so that the visualization can proceed.

3.4 3D High-Interactivity Vocal Folds ABM

The scheduling schemes described in Section 3.3 outperformed other related ABM works in terms of computational throughput (performance will be discussed in Section 6.2.1 and 6.2.2). However, since the computational tasks were distributed among the host and different accelerators, visualization was performed at the end of each iteration after all data were copied back to the host and synchronized. The visualization framerate was thus, bounded by the execution time of the computation and synchronization of the whole iteration.

The scheduling scheme proposed in Section 3.3 was restructured to mitigate

Algorithm 1: Pseudocode describing CPU-GPU scheduling related functions in Driver, Computation and Visualization class

```
1  Function Driver::run():
2      init()
3      launchCPUthreads(2)
4          if thread_id == 0 then
5              Visualization.start()
6          else
7              Computation.start()
8      return

9  Function Visualization::start():
10     while !simulationDone do
11         renderOnGPU()
12         visualizationDone ← 1          // Notify Computation class of
                                             // visualization completion
13         while computationDone ≠ 1 do
               // wait for computation on both CPU and GPUs to
                  complete
14         computationDone ← 0     // reset computation completion flag
15     return

16 Function Computation::start():
17     while !simulationDone do
18         launchCPUthreads(2)
19             if thread_id == 0 then
20                 executeCPUtasks()
21             else
22                 executeGPUtasks()
23         syncAndUpdateWorld()      // Sync CPU and GPU chemical data
24         computationDone ← 1        // Notify Visualization class of
                                             // computation completion
25     return
```

Algorithm 2: Pseudocode describing 3D VF-ABM operations and workflow

```
1  Procedure executeCPUtasks()
       /* model computation                                          */
2      launchCPUthreads(p − N_GPU − 1)
                                          // p denotes the number of
                                              // available CPU cores
3          for each Patch pt ∈ 3Dworld do
4              if pt.conditionMet() then
5                  pt.seedCell()
6              pt.ECMFunction()
7              pt.fragmentECMs()
8          for each Cell c ∈ InflammatoryCells do
9              c.cellFunction()

           /* model update (excluding chemical data update)          */
10         for each Patch pt ∈ 3Dworld do
11             pt.updateECMs()
12             pt.updatePatch()
13         for each Cell c ∈ InflammatoryCells do
14             c.updateCell()

15 Procedure executeGPUtasks()
16     launchCPUthreads(N_GPU)
17         gpu_id ← thread_id
18         if gpu_id == gpu_id_vis then
19             while visualizationDone ≠ 1 do
                   // wait for visualization on GPU to complete
20             visualizationDone ← 0    // reset visualization completion
                flag
21         for each ChemicalType ct ∈ ChemicalTypeSet[thread_id] do
22             diffuseChemicalOnGPU(ct, gpu_id)  // using GPU FFT library
                                                 // (i.e.  NVIDIA cuFFT) for
                                                 // convolution computations
```

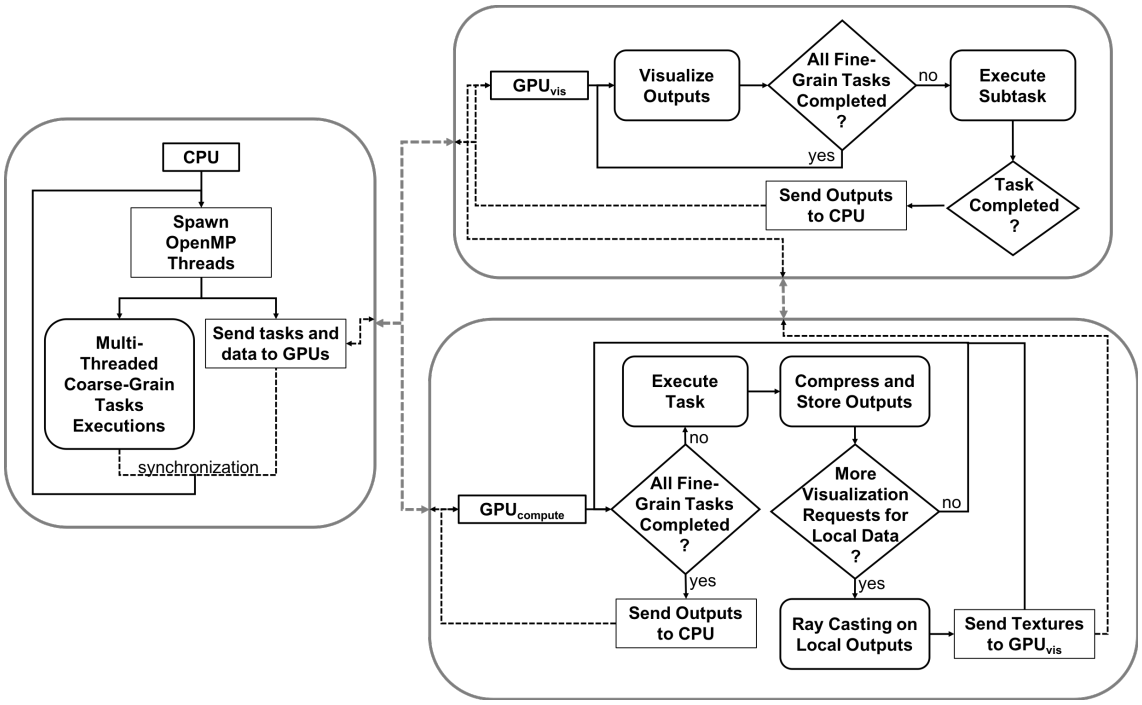


Figure 3.3: Diagram showing work flow and scheduling decision of different tasks on different devices. The coarse-grain tasks gets executed on the host using $p-N_{GPU}-1$ threads, where p denotes the number of available CPU cores. The CPU host communicates with the rendering GPU (GPU_{vis}) to send necessary data for GPU_{vis} to visualize. Between each visualization frame, GPU_{vis} assists with fine-grain computations at a subtask level to lighten the loads of other GPUs ($GPU_{compute}$), since these compute GPUs also assist with volume-rendering visualization subtasks to minimize off-device data transfers.

the bound on user-simulation interactivity by (1) scheduling GPU tasks at a finer level to enable (2) GPU hyper-tasking (GHT) [17]. The benefits of GHT are two folds. First, the GPUs responsible mainly for fine-grain computations ($GPU_{compute}$) can hyper-task between fine-grain computations and ray-casting (visualization subtask). This allows the non-screen-attached GPUs to aid in visualizing their local data and minimize peer communications. Second, GHT allows the rendering GPU (GPU_{vis}) to lighten the computational loads of its peers by executing leftover fine-grain subtasks, while maintaining high level of visualization interactivity.

Fig. 3.3 demonstrates the workflow of the proposed scheduling scheme. This scheme was executed with p CPU threads, where p denotes the number of available CPU cores. A total of $p - N_{GPU}$ threads were used for coarse-grain computations. This part remained unchanged from the original 3D scheduling schemes. In the 3D high-interactivity scheme, the rest of the threads were not simply launching fine-grain tasks, but rather communicating with the rendering GPU (GPU_{vis}) and compute GPUs ($GPU_{compute}$) to orchestrate GHT.

The program used $N_{GPU} - 1$ threads for communications with the GPUs responsible mainly for fine-grain computations ($GPU_{compute}$). Each time a $GPU_{compute}$ completed its fine-grain task, the GPU compressed the output and buffered it for visualization in the next iteration. In addition, these GPUs participated in visualizing their local data from the previous iteration when they finish a task by performing ray-casting locally. Each $GPU_{compute}$ then sent the results as `GLtexture` to the GPU_{vis} for rendering. This prevented the GPUs from having to send large 3D volumetric data to their peer, and only send ready-to-render 2D textures to GPU_{vis}.

One thread was spared for communication with GPU_{vis}. The main responsibility of GPU_{vis} was to perform graphical visualization of all numerical outputs including those residing on the CPU host and other GPUs ($GPU_{compute}$). However, between visualization frames, GPU_{vis} switched to compute mode to assist in fine-grain computations. Since, other GPUs ($GPU_{compute}$) spared some of their resources to assist with the visualization process to reduce amounts of data being sent off-device, it was necessary for GPU_{vis} to lighten the loads of other GPUs whenever it can for optimal performance.

Our tested configuration consisted of one multi-core CPU and two GPUs. However, this scheduling scheme could be extended to work with more than two GPUs by executing the scheduling logic of $GPU_{compute}$ shown in Fig. 3.3 on additional GPUs. The added GPUs would be beneficial if the execution time of the fine-grain tasks on GPUs exceeds that of the CPU coarse-grain tasks. This scheduling scheme would thus, scale out with additional GPUs as the amount of fine-grain tasks increases.

Chapter 4: Numerical Simulation Optimizations

A challenge in biological simulation is to handle the differences in spatiotemporal scales between cellular and chemical interactions [79]. To address this challenge, we designed a mechanism to compute the processes with finer spatiotemporal steps (chemical diffusion) at the scale comparable to the cellular level processes. Cellular functions and chemical diffusion were then optimized separately for execution on multi-core CPU and GPU, respectively.

4.1 Cellular Functions

Cellular level processes such as cell migration and protein synthesis are complex, but relatively less data-intensive compared to chemical diffusion. Thus, these functions were optimized using OpenMP to be executed on the CPU cores. This section describes two main optimization approaches used to fully leverage parallelism on a shared memory multi-core device: concurrent containers and parallel update mechanisms.

4.1.1 Concurrent Cell Container

The program maintains pools of biological cells (ABM agents). The agent pools can expand or shrink due to cell proliferation and cell deaths. The C++ `std::vector` was originally used to maintain the pools of biological cells in the sequential version. However, when cell related functions were parallelized, this container of choice became the bottleneck of the program. There exists a `concurrent_vector` class which is thread-safe, unfortunately, there is no support for parallel deletion making the deletion of dead agents nonparallelizable. To alleviate this limitation, a custom data structure, `ArrayChain` was developed. The goal is for this new thread-safe container to manage an agent pool with concurrency and perform all agent-related operations in $O(1)$ on average. The concurrency was achieved without the need for explicit mutual exclusion through the requirements that the set of agents accessed by each thread does not overlap with that of any other threads. The following are the details of this new structure.

```
template<AgentType T>
struct ArrayChain
{
        T* data[MAX_ARR_SIZE];
        vector<int> freeLists[NUM_THREADS];
        struct ArrayChain* next;
        int numFreeApprox;
```

```
}
```

Listing 4.1: Code snippet for definition of nodes in ArrayChain

Members:

- `T* data[MAX_ARR_SIZE]` stores the actual data (cell pointers)
- `vector<int> freeLists[NUM_THREADS]` stores lists of indices of free slots in data. Each vector contains best effort information local to each thread. `freeLists[tid]` only contains indices of the slots in data that has been emptied out by thread with id `tid`. Thus, each list or even all of them combined may not be exhaustive.
- `struct ArrayChain*` next stores a pointer to the next node in the chain. `NULL` if this is the last node.
- `int numFreeApprox` stores a number of contiguous elements at the end of the list. This is used when threads request for parallel batch insertion to data. For example, in Fig. 4.1, there are 589 slots at the end of data that are free and contiguous, thus parallel batch prefix sum insertion of total number of elements of at most 589 can be performed in node 2 of the chain.

Agent Insertion: Let's say thread *tid* would like to add a new agent to the list. It would call `insert()`, described in Algorithm 3 in an attempt to add to a recycled slot (from its `freeList`). If `insert()` returns `true`, then this new agent was successfully added. If it returns `false`, then thread *tid* will attempt to add this new agent (and other agents it has not added) through a participation in parallel

prefix insertion with other threads.

Algorithm 3: ArrayChain single-element insertion and deletion algorithm

```
1  Procedure insert(ArrayChain agentList, Agent newAgent)
2      for each ArrayChainNode arrNode ∈ agentList do
3          if ¬arrNode.freeList[tid].isEmpty() then
               // empty recycled slot found
4              index ← arrNode.freeList[tid].pop()      // get index of free
                slot
5              arrNode.data[index] ← newAgent                  // add agent
6              return true
7      return false
8  Procedure delete(ArrayChain agentList, index)
9      nodeIndex ← getNodeIndex(index)
10     nodeOffset ← getNodeOffset(index)
11     arrNode ← agentList.head
12     for counter = 1 to agentList.size() do
13         arrNode ← arrNode.next
14     arrNode.data[nodeOffset] ← NULL
15     addNode.freeList[tid].push(nodeOffset)
```

Prefix sum insertion happens when at least one thread ran out of local free indices. When this happens, the program will perform prefix sum on the array

containing number of elements each thread would like to add to the list. Once the prefix sum is performed, each thread will then have the correct offset to data in the last node of the chain so they can all insert their elements in parallel.

If numFreeApprox is less than total number of things to add, then:

- Add number from `(MAX_ARR_SIZE - 1 - numFreeApprox)` to `(MAX_ARR_SIZE - 1)` to `freeLists[0]`
- Allocate new node
- Add all elements to data in the new node in parallel

Notice that to insert a single agent, if a thread finds an empty slot through its `freeList` the thread does not have to wait for any other thread (full concurrency) and the operation complexity is $O(1)$. If the thread has to participate in a prefix sum parallel insertion, the complexity is still $O(1)$, concurrency is still maintained, however, the prefix sum computation becomes the execution barrier. The prefix sum computations themselves are constant as the complexity is proportional to the number of CPU threads. The worst case is when it has to iterate through all the nodes in the `ArrayChain`, the complexity becomes $O(m)$, where m is the number of nodes which is roughly $\frac{n}{MAX\_ARR\_SIZE}$, assuming constant size nodes.

Agent Deletion: Deletion of a single agent takes clearly constant time with full concurrency, as it simply frees the agent space and marks entry `NULL`, then adds the entry index to the `freeList`.

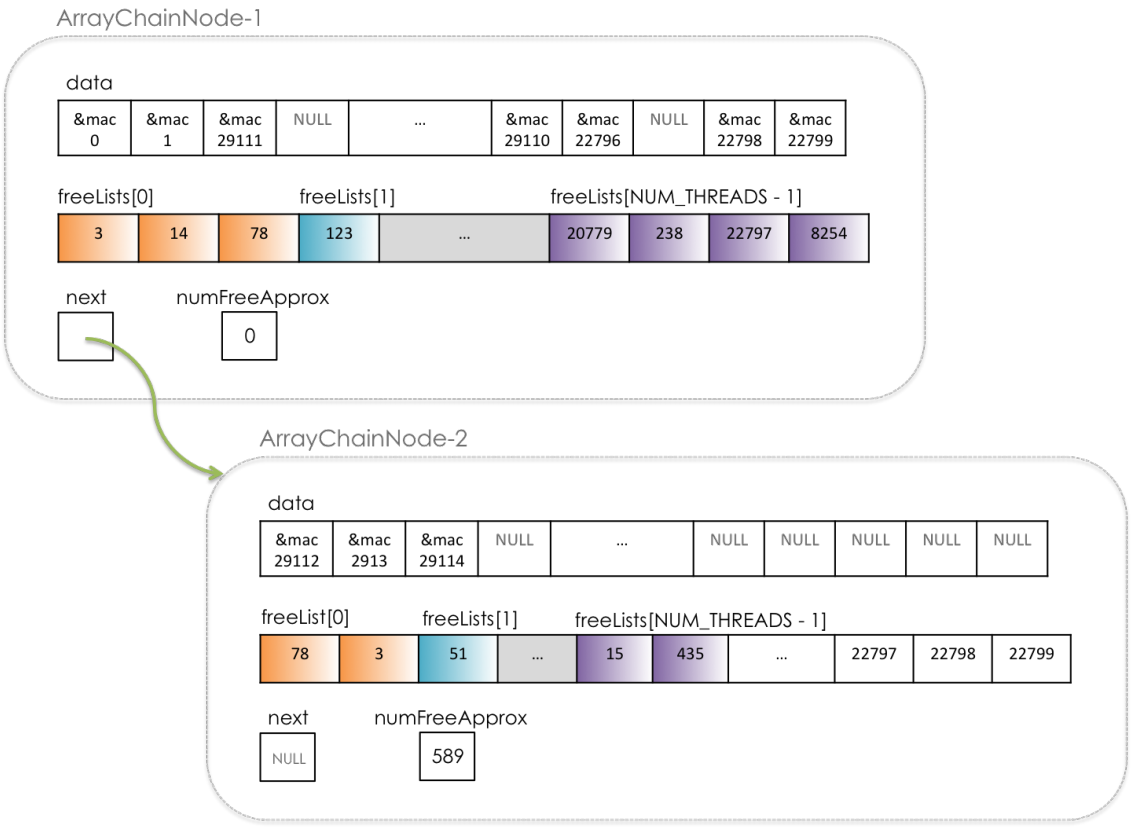


Figure 4.1: An example of ArrayChain of macrophages (mac) with two nodes.

4.1.2 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. Consequently, 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 the simulation progresses, 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 that 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 4. 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.

Algorithm 4: Agent Atomic Move

Input: P = target patch

Output: $rc = true$ if move was successful, $false$ otherwise

```
1 if ¬atomic_test_and_set(P.isoccupied) then
2     this.x ← P.x;
3     this.y ← P.y;
4     return true;
5 else
6     return false;
```

4.2 Chemical Diffusion

Chemical diffusion was the most demanding computational component of the VF ABMs. As previously mentioned, its computational demand was primarily a result of the extremely small spatiotemporal scale and high rate at which chemical diffusion occurs. To reduce the computational load, a convolution-based method was used to simulate the diffusion process [14]. A Fast Fourier transform (FFT) was then used to reduce the complexity of convolution computations. Lastly, kernel size reduction was achieved by extracting the most dense segment of the Gaussian kernel to optimize the diffusion performance [16]. Note that, since we deal with regular grids for the ABM world, finite difference method (FDM) is used as opposed to the more computationally intensive integral schemes.

4.2.1 Discrete Finite Scheme

Diffusion equation with decay in 2D can be written as follows:

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right) - \gamma c, \tag{4.1}$$

where c is the chemical concentration, D is the diffusion coefficient and γ is the decay constant. For the 3D case, the diffusion equation with decay can be extended as

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2} + \frac{\partial^2 c}{\partial z^2}\right) - \gamma c, \tag{4.2}$$

Assuming that $\Delta x = \Delta y = \Delta z$, and using a Taylor expansion to discretize the continuous 3D diffusion equation, we get

$$\begin{aligned} c\left(x, y, z, t + \Delta t\right) = & \left(1 - \frac{4D\Delta t}{\Delta x^2} - \gamma \Delta t\right) c\left(x, y, z, t\right) + \\ & \frac{D\Delta t}{\Delta x^2}\left[c\left(x + \Delta x, y, z, t\right) + c\left(x - \Delta x, y, z, t\right) + \right. \\ & c\left(x, y + \Delta y, z, t\right) + c\left(x, y - \Delta y, z, t\right) + \\ & \left. c\left(x, y, z + \Delta z, t\right) + c\left(x, y, z - \Delta z, t\right)\right] \end{aligned} \tag{4.3}$$

Using Von Neumann Stability Analysis method to study the growth of the

waves e^{ikx} [193], we have the following stability conditions:

$$\Delta t \leq \frac{\Delta x^2}{6D}. \tag{4.4}$$

4.2.2 Convolution Based Diffusion

As shown in Table 4.1, the largest value of D in the set of chemical types in VF ABM is $900\frac{\mu m^2}{minute}$ [194], with patch width $\Delta x = 15\mu m$. The condition $\Delta t \leq 2.5$ s needs to hold to meet stability constraints. Clearly, the complexity of the simulation using discrete finite scheme would be unnecessarily high if the model evolved at $\Delta\tau = 2.5$ s rather than $\Delta\tau = 30$ min or 1800 s.

By letting $\lambda = \frac{D\Delta t}{\Delta x^2}$, Eq (4.3) can be rewritten as

$$\begin{aligned} c\left(x,y,z,t+\Delta t\right) = {} & \left(1-6\lambda-\gamma\Delta t\right)\cdot c\left(x,y,z,t\right) \\ & \lambda\cdot c\left(x+\Delta x,y,z,t\right)+\lambda\cdot c\left(x-\Delta x,y,z,t\right)+ \\ & \lambda\cdot c\left(x,y+\Delta y,z,t\right)+\lambda\cdot c\left(x,y-\Delta y,z,t\right)+ \\ & \lambda\cdot c\left(x,y,z+\Delta z,t\right)+\lambda\cdot c\left(x,y,z-\Delta z,t\right) \end{aligned} \tag{4.5}$$

or,

$$c\left(x,y,z,t+\Delta t\right) = \sum_{i=x-1}^{x+1}\sum_{j=y-1}^{y+1}\sum_{k=z-1}^{z+1} c\left(i,j,k,t\right)\cdot f\left(x-i,y-j,z-k\right), \tag{4.6}$$

where

$$f(x,y,z) = \begin{cases} 1-6\lambda-\gamma\Delta t & x=0 \bigwedge y=0 \bigwedge z=0 \\ \lambda & x=\pm 1 \bigwedge y=0, \bigwedge z=0, or \\ & y=\pm 1 \bigwedge x=0, \bigwedge z=0, or \\ & z=\pm 1 \bigwedge x=0, \bigwedge y=0 \\ 0 & otherwise. \end{cases}$$

Clearly, Eq (4.3) is equivalent to Eq (4.6), thus $c(x,y,z,t+\Delta t) = c(x,y,z,t) * f(x)$, where $*$ represents convolution. This results in the ability to fast forward this process to capture diffusion at a large time step, $\Delta\tau$, without violating stability constraints using convolution.

To compute $c(x,y,z,\tau+\Delta\tau)$, where $\Delta\tau = m \cdot \Delta t$, the chemical concentrations from previous step, $c(x,y,z,\tau)$, was convolved with $f(x,y,z)$, m times. The commutative property of convolution implies that convolving $f(x,y,z)$ with itself m times results in $f_m(x,y,z)$, and the diffused concentrations at each iteration can be computed as

$$c(x,y,z,\tau+\Delta\tau) = c(x,y,z,\tau) * f_m(x,y,z). \tag{4.7}$$

The diffusion computation can thus be accelerated by computing Eq 4.7 at a large time step, $\Delta\tau$, without violating stability constraints. The effective diffusitivity of IL-1β in tissue, for example, is 900 $\frac{\mu m^2}{min}$ [194]. In a 15 μm patch world, a 30-minute time step implies that the program has to calculate $c(x,y,z,\tau) * f_{720}(x,y,z)$ at each

time step. In other words, a chemical on a given patch (x,y,z) has a spatial diffusion range of $x\pm720$, $y\pm720$ and $z\pm720$, within a window of dimension $1441\times1441\times1441$, which covers approximately 3 billion 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 folds modeling, our tissue area of interest has epithelium on the outermost layer. This means we have effectively one wall, or 1-side 0-flux boundaries. And the rest of the walls can be padded with empirically obtained baseline chemical levels, or constant padding. The convolution computation was implemented using cuFFT [195], an NVIDIA CUDA FFT library, to speedup the runtime.

Effective Diffusitivity ($\mu m^2/minute$)							
TNF-α	**TGF-β1**	**FGF**	**MMP8**	**IL1**	**IL6**	**IL8**	**IL10**
900	780	780	780	900	810	900	900

Table 4.1: Effective diffusion coefficients used in VF ABM. TNF-α, TGF-β1, IL1 and IL6 values were taken from [194].

4.3 Kernel Reduction

The diffusion kernel was computed by convolving the initial coefficient function, $f(x, y, z)$, in Eq (4.6), with itself $m = \Delta\tau/\Delta t$ times, where $\Delta\tau$ is the biological time step of 30 *minutes* and $\Delta t = \Delta x^2/6D$ is the diffusion time step subjected to

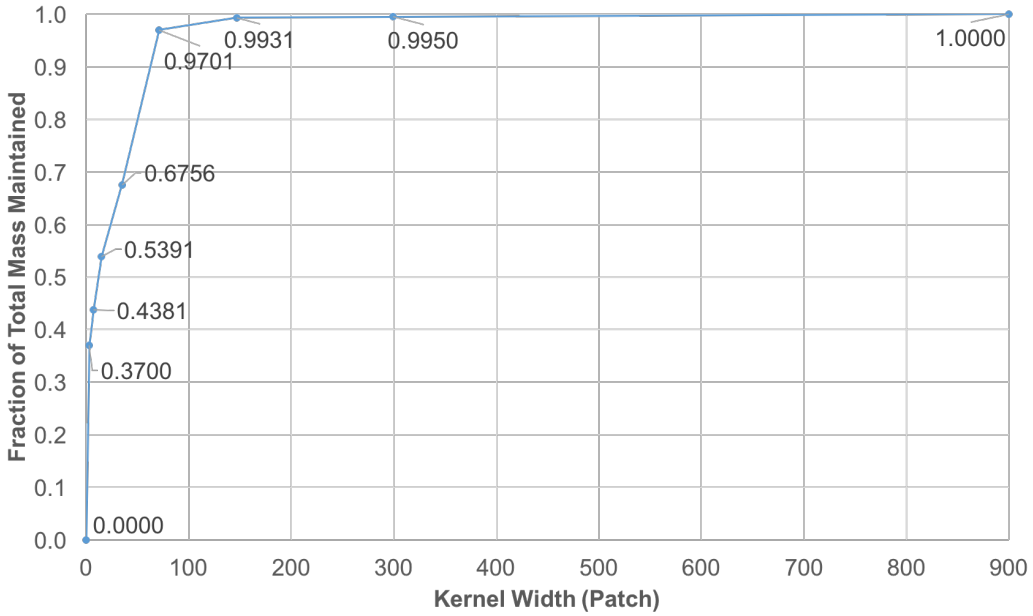


Figure 4.2: Diffusion kernel reduction mass vs. kernel width. This plot shows mass coverage with respect to extracted window width. The size of each kernel is $width^3$ patches. It is observed that by cutting down the size from 1441^3 down to 147^3, only a fraction of 0.0061 of the mass is lost in each iteration.

the stability constraints (Eq 4.4). As calculated earlier, the effective diffusitivity of IL-1β of 900 $\frac{\mu m^2}{min}$ results in a $1441 \times 1441 \times 1441$ kernel.

Note that $f(x, y, z)$ is smoother as it gets convolved with itself, thus a Gaussian shaped diffusion kernel is obtained. The values in Gaussian distributions are highest at the center. These values decrease and approach *zero*, the further they are from the center. This observation enabled the reduction of the kernel size by focusing on the center window, while keeping almost 100% of kernel mass. The coverage levels of the kernel mass with respect to extracted window sizes are plotted in Fig 4.2.

Chapter 5: Visualization Optimizations

5.1 In Situ Visualization Protocol

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) [192]. 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 does not exhibit neither property 2 nor 3. Lastly, the *client-render remote simulation* scheme may manifest both 2 and 3; however, it does not fully take advantage of the powerful server since it redirects the rendering to the client.

The In Situ Remote Visualization paradigm [142] 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 that gives any Unix or Linux remote display software the ability to run OpenGL applications with full 3D hardware acceleration [196]. Figure 5.1 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.

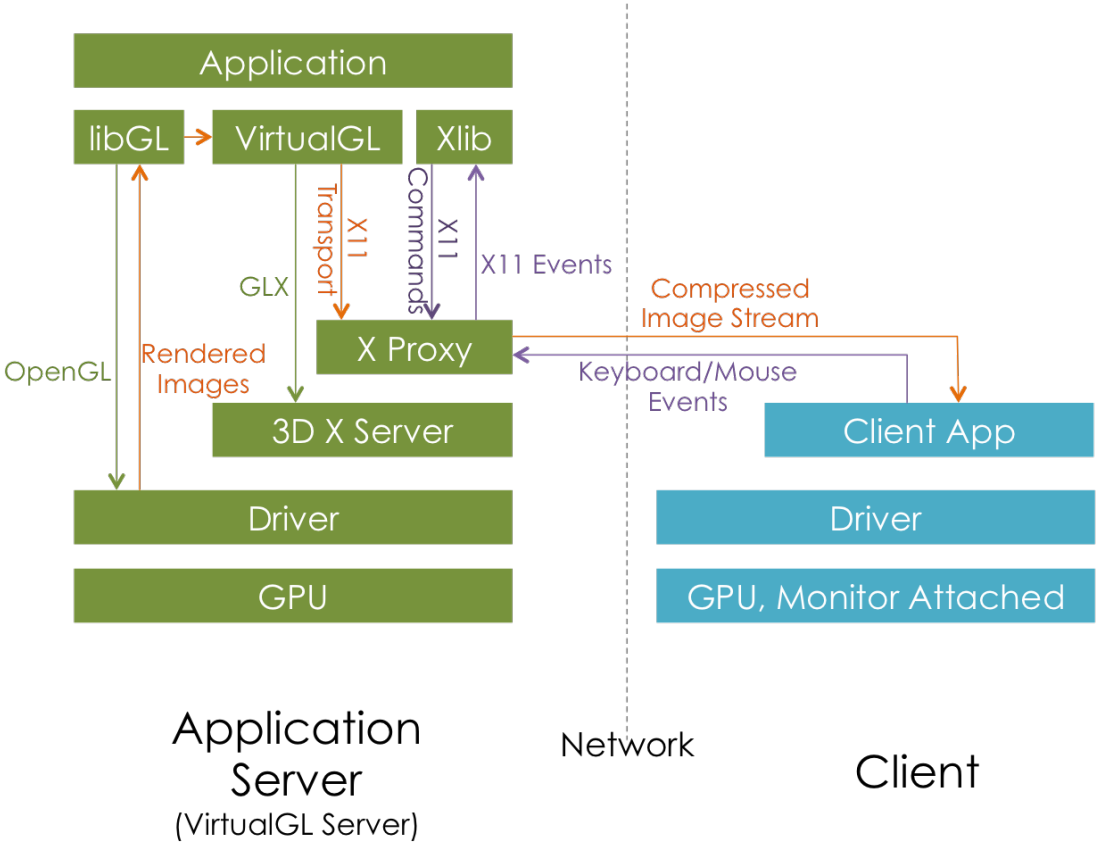


Figure 5.1: Diagram depicting the system configuration for *In Situ* remote visualization using X11 transport with an X proxy.

5.2 Data Access Pattern Observations and Categorizations

For optimization purposes, we categorize a dynamic pattern of a data type based on two properties. First, a data access pattern is defined by the distribution characteristics in a single iteration. For example, the distribution of a chemical concentration is typically smooth due to diffusion, whereas ECM protein and cell distributions exhibit some degree of discontinuity. Second, the dynamics are defined by how the distribution changes from one iteration to another, for example localized versus distributed changes. Table 5.1 shows the categorization of dynamic patterns for the different types of output and which device they were computed on.

	Smooth	**Discontinuous**
Localized	0—n/a	I—ECM Proteins
Distributed	II—Chemicals	III—Cells
	GPU	**CPU**

Table 5.1: Data dynamic pattern categories

Notice that none of the patterns observed in wound healing ABMs fall into category 0. Data type that are discontinuous and distributed fall into category III. Examples include mobile agents such as neutrophils, macrophages and fibroblasts, or non-mobile agents such as platelets. In this work we focused our optimization efforts only on protein (signaling chemicals and ECM) data that fall into category I and II.

Data category I are quantities that do not diffuse or spread, thus changes are typically local with discontinuous distributions. These are mostly structural

quantities. For example, ECM proteins form an underlying structure of the tissue and provide building blocks of the overall structure similar to laying out bricks for a building. Once a brick is laid, it stays there until it is degraded or destroyed. In wound-healing models, these quantities are localized, and thus changes to these quantities happen mostly in highly-active areas such as wound sites.

In contrary, category II data diffuse or spread, thus their distributions are typically smooth. Once the mass gets deposited at a location, it undergoes immediate changes in concentrations and reaches other areas. This includes diffusible quantities such as chemicals, gas, and heat. The dynamics of these quantities are very distributed in nature. The changes usually keep happening and affect other areas until an equilibrium is reached. However, in wound healing models, inflammatory cells may not stop protein secretion until they return to homeostasis [197]. Thus, an equilibrium may not be reached until the end of the simulation, resulting in constant changes all over the simulation area.

5.3 Data Access/Update Minimization Technique

5.3.1 Data Category I

As discussed in the previous section, this type of data involves structural change and the outputs reside on the CPU host. These data, thus, need to be copied from CPU to GPU_{vis} during visualization. Structural data change locally i.e. the change in one location does not directly impact other locations. More specifically, in wound healing models, most activities occur in the damaged areas.

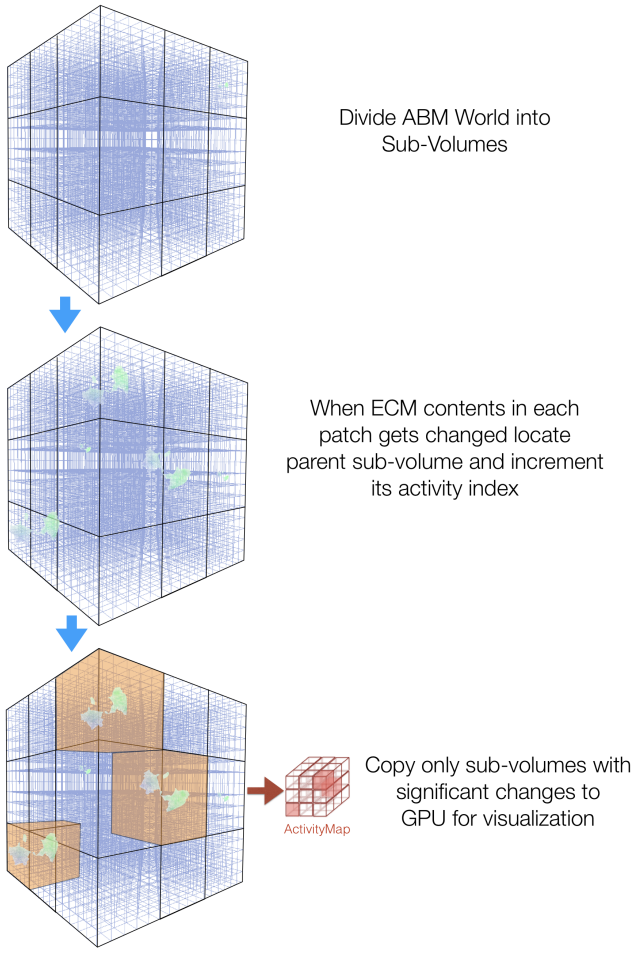


Figure 5.2: An example sequence demonstrating the Host-Device Activity-Aware Data Copy (HADC) technique to minimize the transfer of redundant category I data. First, the 3D ABM world gets divided up into smaller sub-volumes. In each iteration, when a patch is modified, the parent sub-volume gets identified. The corresponding entry in the activity map gets marked as 'dirty'. Only the 'dirty' sub-volumes get copied to the GPU for visualization.

Given these observations, we developed a Host-Device Activity-Aware Data Copy (HADC) technique to reduce the copying of redundant data.

5.3.1.1 HADC Algorithm

Fig. 5.2 demonstrates the work flow of HADC during the simulation. First, the ABM world is divided into smaller sub-volumes. In each iteration, when a patch gets modified, the program identifies the parent sub-volume of that patch. The algorithm then looks up the corresponding entry in the activity map and increments its activity index. During the data buffering stage, only the data in the sub-volumes that had gone through significant changes get copied to the GPU for visualization. In this way, the amount of redundant data copies is minimized, resulting in a faster buffering process.

5.3.1.2 HADC Sub-Volume Size Recommendation Heuristic

The size of the sub-volumes is a very important factor in determining the performance of HADC. If the size is too large, a large amount of unnecessary data copies will be performed. However, if the size is too small, since each small sub-volume will be copied in a separate transaction, the latency of host-device copy operations will hurt the overall performance. To find an optimal partition, the program runs the model, and logs the locations of activities in each iteration. For a given M points (representing activity locations) in a log from the most active iteration, the goal is to find a partition of size $w \times h \times d$, where the M points appear in as few

partitions as possible. In other words, the objective is to minimize the partition size (space) and number of communications (time) between host and device, where the two extremes are $< O(WHD)space, O(1)time >$ and $< O(1)space, O(M)time >$.

To determine a good partition size, a modified octree algorithm is used. This heuristic consists of two passes: top-down and bottom-up. At the end of each pass, each node (partition) will either be labeled white (insignificantly low density) or black (significantly high density). The first pass starts top-down from the whole simulation grid, and split on the density conditions given the parameters ρ_{low} and ρ_{high}. Once the program is done splitting as all nodes satisfy one of the density conditions (node density $< \rho_{low}$ or $> \rho_{high}$), the program will return a list of black node dimensions. Here, the smallest volume dimensions will be picked to minimize the partition size. The program then execute the second pass in an attempt to combine consecutive partitions if and only if the density condition requirements still hold. This bottom-up pass is executed to further minimize the number of host-device communications.

Given a wound healing model $m =< wc, mc, ic, r >$, where wc denotes the wound configuration (dimensions, wound location), mc denotes model configuration (dimensions), ic denotes the initial conditions (patients cytokine levels, treatment type etc.) and r denotes the model rules, if we know the optimal subvolume size of m, then we can choose a scaled subvolume size for model $m' =< wc', mc, ic', r >$. For example, if the wound depth of m and m' is 1 mm and 0.5 mm respectively in the x-direction, then we can scale the optimal subvolume size of m in the x direction by $\frac{1}{2}$ and reuse it for m'. This way, once we determined an optimal subvolume size

for model m, we can apply it to any model m' defined earlier.

5.3.2 Data Category II

The computations of diffusible quantities involve relatively large amounts of data, thus they were performed on GPUs. These output data are, thus, local to the GPUs. Two approaches were used to reduce data access: host-side sampling and device-side sampling. The host-side sampling minimizes number of data points accessed before the OpenGL rendering functions get called through constant and adaptive sampling.

To lower the amounts off-device data copies, the device-side sampling scheme compressed and store computed data on the producer GPU using linear sampling so each GPU can assist in ray-casting. The details of both schemes will be described in the following sections. The performance improvement and accuracy trade-offs are discussed in Section 6.3.

5.3.2.1 Host-Side Sampling

The first host-side sampling approach is simply constant sampling. The world environment is divided into tiles of size $grid_x \times grid_y \times grid_z$, where the corresponding grid point represents the values of the tile centered at the point and within the radius of $grid_x/2$, $grid_y/2$ and $grid_z/2$ in the x-, y-, and z-dimension, respectively. This naive approach was used to improve the visualization speed in our earlier work [16]. At the perspective of the entire simulated volume, the appearances of

sampled simulation were indistinguishable for up to $(6 \times 6 \times 6)$-segment sampling. However, the output became pixelated when the view is magnified and the camera focuses on a smaller area. In particular, the wound area was not rendered with reasonable fidelity.

Adaptive sampling is used to optimize the data access while enhancing the resolution of the visual output in important areas. This sampling scheme helps keep the execution time low, and yet the details in the output presented are not compromised. The details of execution time and visualization outputs will be discussed in Section 6.2.2.2.

For models aiming to capture injury undergoing inflammation and repair processes, the wound site is the most active area. Therefore, the highest importance index was assigned to the wound site volume. The margin around the wound site, and the rest of the tissue are then respectively assigned less and least importance. The adaptive sampling pipeline diagram in Fig 5.3 illustrates data processing steps used to sample and send data to the visualization pipeline. The program, by default, divides the whole tissue volume into three sections by first inspecting the initial wound position and size, followed by adding a margin around the wound based on user inputs, and then labeling this volume as the most active (region 1). The program lets the user specify the volume ratio between medium- and low- activity area, and splits the rest of the volume into regions 2 and 3 accordingly. As our visualization intends to highlight wound activity, the re-sampling is performed only once, thus the memory footprint is not heavily affected by the re-sampling process.

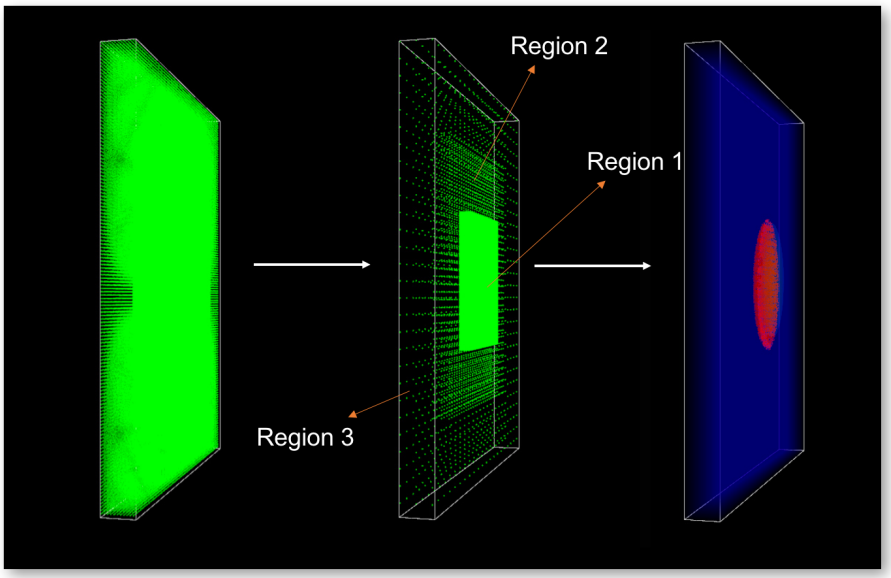


Figure 5.3: Adaptive sampling pipeline. From the whole set of data (left), the data are read with resolution conforming to the importance index (middle). The sampled data are then processed and sent to the visualization pipeline to be rendered (right).

5.3.2.2 Device-Side Sampling

In multi-scale simulations, the diffusion is a fine-grain process and is most suitable to be executed on GPUs. Thus, the data needed by the visualization would have been temporarily stored in the GPU memory at some point. An ideal situation would be to save the diffusion output on the GPUs. Storing data on the GPU will make the data available for visualization throughout each given iteration. However, the size limitation of GPU memory may not allow the storage of all data points in large data simulations.

To avoid off-device data visualization, the buffer size needs to be reduced to fit on the GPU. Here, the data size is reduced through a simple constant sampling process. The output gradient details are then preserved by taking advantage of the GPU

texture memory. Through the CUDA texture filter mode, `cudaFilterModeLinear`, a sample fetched from a CUDA texture is automatically interpolated [198]. This functionality of texture memory helps preserve the temporal dynamics of the output by filling in the gaps between sampled points with interpolated values. Note that CUDA does not allow write operations through a texture reference. Thus, sampling from GPU memory to a texture will need to be performed via a surface reference bound to the same CUDA array.

One may argue in favor of more complex data sampling and interpolation methods. However, as those methods improve the accuracy of the sampled data, they also add complexity. In large data simulations, millions to billions of data points are being visualized in each frame. With such large amounts of data points, the added accuracy may not be visible. Thus, we studied the image quality versus execution time and report our findings in Section 6.3. It appeared that the simple sampling with the help of CUDA texture produced visualization that looked indistinguishable to the original data and maintained acceptable accuracy of 95%. Thus, we did not deem it necessary to add more complexity to the program in this context.

Chapter 6: Results

To evaluate the techniques proposed in previous chapters, this chapter begins with a section describing configurations of 2D, 3D and highly-interactive 3D VF ABMs. The performance improvements gained using different sets of techniques described in Chapter 3, 4, and 5 are then discussed for each respective model. Next, the chapter studies the performance-accuracy trade-offs for sampling techniques described in Section 5.3.2. Finally, the chapter ends with the details on model verification.

6.1 Model Configurations

We have developed VF ABM models for two different mammals: rat (3D) and human (2D and 3D). The rat 3D VF ABM serves as a test model due to its size and the availability of empirical data. The model configurations were determined based on empirical data and vocal fold literature reviews [108, 199–204]. Table 6.1 lists the scale of the 2D and 3D VF ABMs implemented in this work. In addition, the details of the set of techniques used in each model is listed in Table 6.2. These tables serve as references in terms of simulation scales and optimizations for performance evaluation that will follow in later sections.

Item	Unit	2D Human	3D Rat	3D Human
World				
Size	$patches^3$	1660 x 1160 x 1	71 x 200 x 142	1390 x 1006 x 110
	mm^3	24.9 x 17.4	1.4 x 1.0 x 0.5	20.85 x 15.09 x 1.65
Patch size	μm^3	15 x 15 x 1	7 x 7 x 7	15 x 15 x 15
Total number of patches	10^6 units	1.9	2.0	154
ECM data	types	3	3	3
	10^6 data points	5.8	6.0	461
Chemical data	types	8	8	8
	10^9 data points	0.015	0.016	1.2
Inflammatory cells (initial)				
Neutrophils	10^6 cells	0.023	0.0005	1.72
Macrophages	10^6 cells	0.023	0.0003	0.97
Fibroblasts	10^6 cells	0.184	0.0035	12.20
Simulated time-step	minutes	30	30	30

Table 6.1: Summary of 2D and 3D VF ABM configurations

6.2 Performance Evaluation

6.2.1 2D Human Model

6.2.1.1 Computation Only Performance

Different versions of 2D VF ABM were implemented for performance evaluation purposes as shown in Table 6.3. These versions follow the same model rules, but differ in computing resource utilization. They were tested and benchmarked on a compute node with 16-core Intel(R) Xeon(R) E5-2690 CPU and NVIDIA Tesla K20c GPU.

As shown in Figure 6.1, the GPU-mCPU-Overlap implementation achieves the

	Section	2D	3D	High-Interactivity 3D
Fine-grain/coarse-grain overlap	3.2	✓		
Fine-grain/coarse-grain/vis overlap	3.3		✓	
Fine-grain/coarse-grain/vis overlap with GHT	3.4			✓
OpenMP parallelization	4.1	✓	✓	✓
FFT convolution-based diffusion	4.2	✓	✓	✓
Kernel reduction	4.3		✓	✓
In situ visualization	5.1	✓	✓	✓
HADC	5.3.1			✓
Host-side sampling	5.3.2.1		✓	
Device-side sampling	5.3.2.2			✓

Table 6.2: Summary of techniques used in each VF ABM implementation

	Tasks Executed on		
Implementation	Single-core CPU	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 6.3: Summary of 2D VF ABM implementations

best performance. This implementation follows the techniques discussed in Section 3.2, 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:

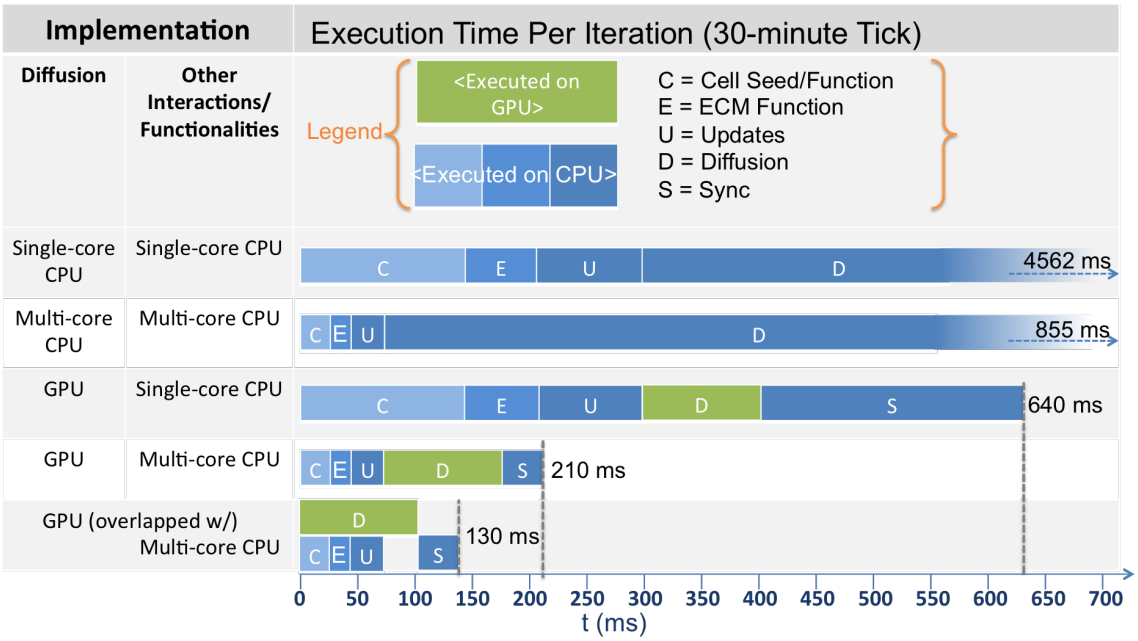


Figure 6.1: Performance of 2D Vocal Fold Inflammation and healing ABM on different processing platforms.

$$t_{total} = max\{t_{CPU_{maxthreads}}, t_{GPU_{maxthreads}}\} + t_{sync}, \tag{6.1}$$

where $t_{CPU_{maxthreads}}$ and $t_{GPU_{maxthreads}}$ 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 $t_{device1_{maxthreads}} \geq t_{device2_{kthreads}}$, 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 aforementioned compute node, $t_{GPU_{maxthreads}} \geq t_{CPU_{8threads}}$, thus the load is most balanced when executing with 8 CPU threads.

Our implementations are able to execute the 2D VF ABM at a scale that is

infeasible on a popular existing ABM framework, NetLogo [205]. To demonstrate the performance gain of the proposed techniques compared to an existing ABM framework, we obtained 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 ran 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 **400x** speedup.

Despite differences in underlying hardware, D'Souza's work on Tuberculosis (TB) ABM Simulation [75] was arguably most suitable for performance comparison with the work reported in this paper. The aforementioned TB ABM described a complex multi-scale biological system of agents that communicate via chemical signals, which aligned 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 consisted of 30x world size with 1000x the number of initial cells, and took only 25 seconds, i.e. 20x less, to perform a 4-day simulation.

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

and heavy memory traffic.

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

Table 6.4: Performance comparison of various 2D VF ABM implementations

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

Table 6.5: Average execution time of remote *in situ* 2D VF simulation

6.2.1.2 Computation-Visualization Performance

The GPU-mCPU-overlap computation implementation, which shows the best performance from section 6.2.1.1 was coupled 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 6.2, 6.3) was 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 6.5, 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

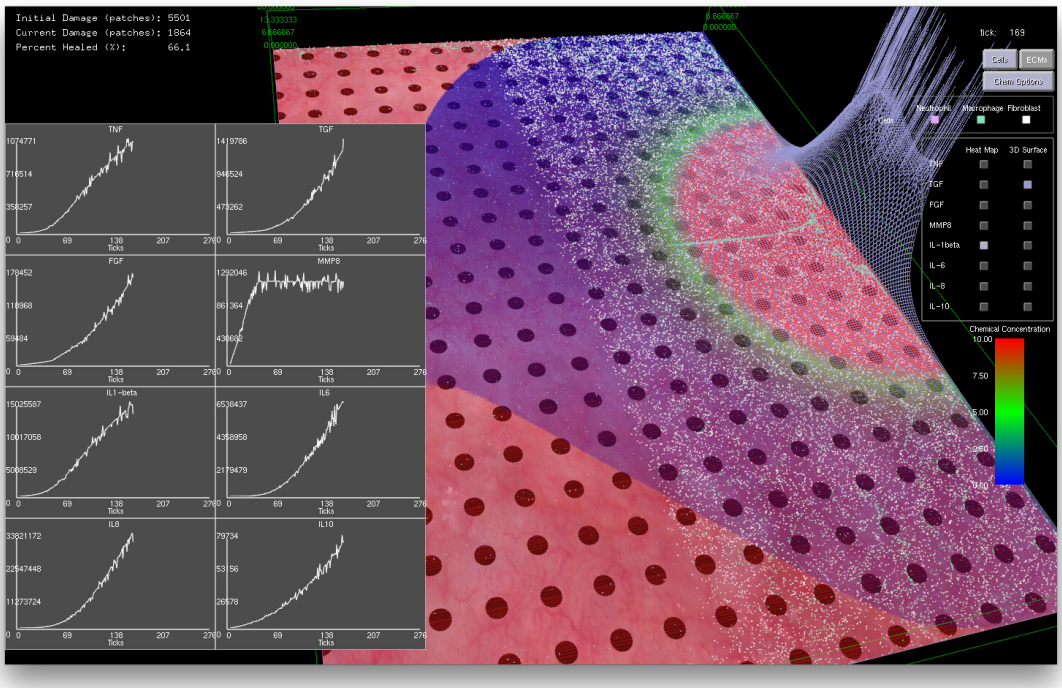


Figure 6.2: A screenshot of a running 2D human vocal fold inflammatory and wound-healing process with aggregated chemical statistics plots and chemical visualization on (heat map and surface plots)

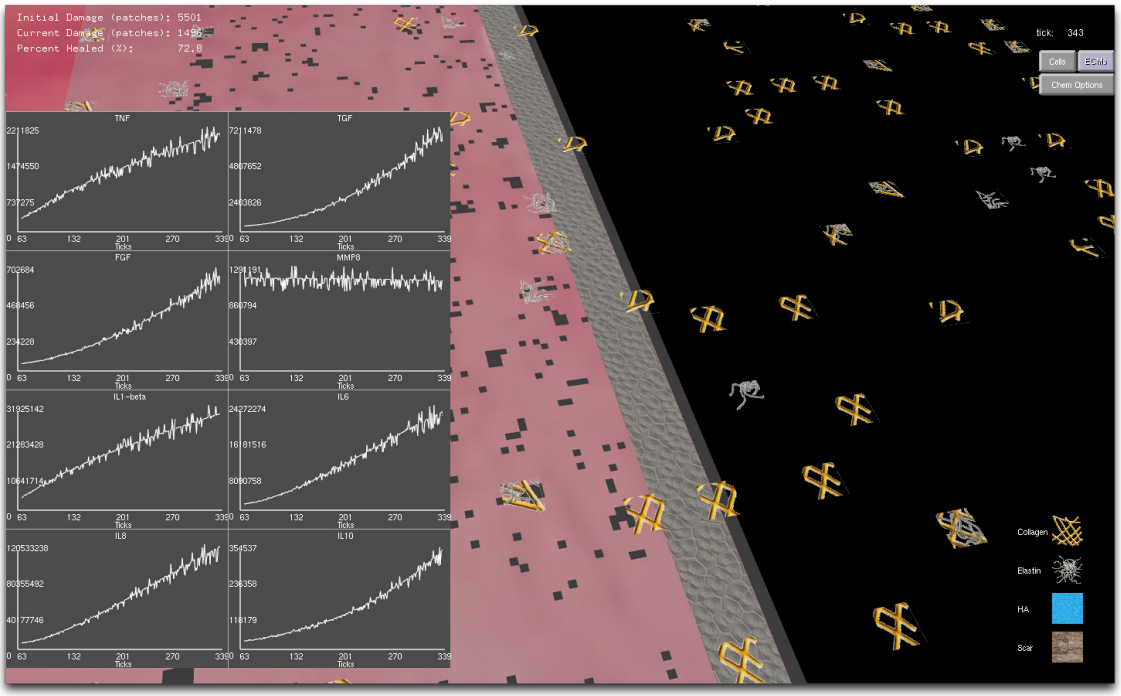


Figure 6.3: A screenshot of a running 2D human vocal fold inflammatory and wound-healing process with aggregated chemical statistics plots and ECM visualization on

with very minimal overhead. Therefore, the total time to simulate, visualize, and render outputs on the client terminal can be kept under 200 ms per iteration.

6.2.2 3D Human Model

6.2.2.1 Computation Only Performance

Due to the efficiency of the FFT-based diffusion method, diffusing 1.2 billion point 3D chemical data on two GPUs only takes 2.5 s per iteration. Since the set of cellular coarse-grain tasks take about 4 s to execute, these tasks become a performance bottleneck. That is, the time the VF ABM takes to complete the computation component of one single iteration depends on how long it takes to execute the cellular tasks plus time to synchronize results. Fig 6.4A shows the execution time for the compute component as a function of number of CPU threads overlapping with two GPUs. These results indicate that the best performance using 32 threads takes approximately 6.2 s per iteration on average. The average speedup of the computation component as well as compute-task speedups across 240 iterations over different numbers of threads are plotted in Fig 6.4B. Tasks are grouped into model function (cell/ECM/synchronization) and update routines, and their speedups within each respective group were then averaged. Notice that the update tasks which consist of mostly memory access operations are memory bound, thus showing poor scalability. In contrast, other model function tasks involve more computation, thus these tasks show good scalability, making the overall speedup of the program satisfactory.

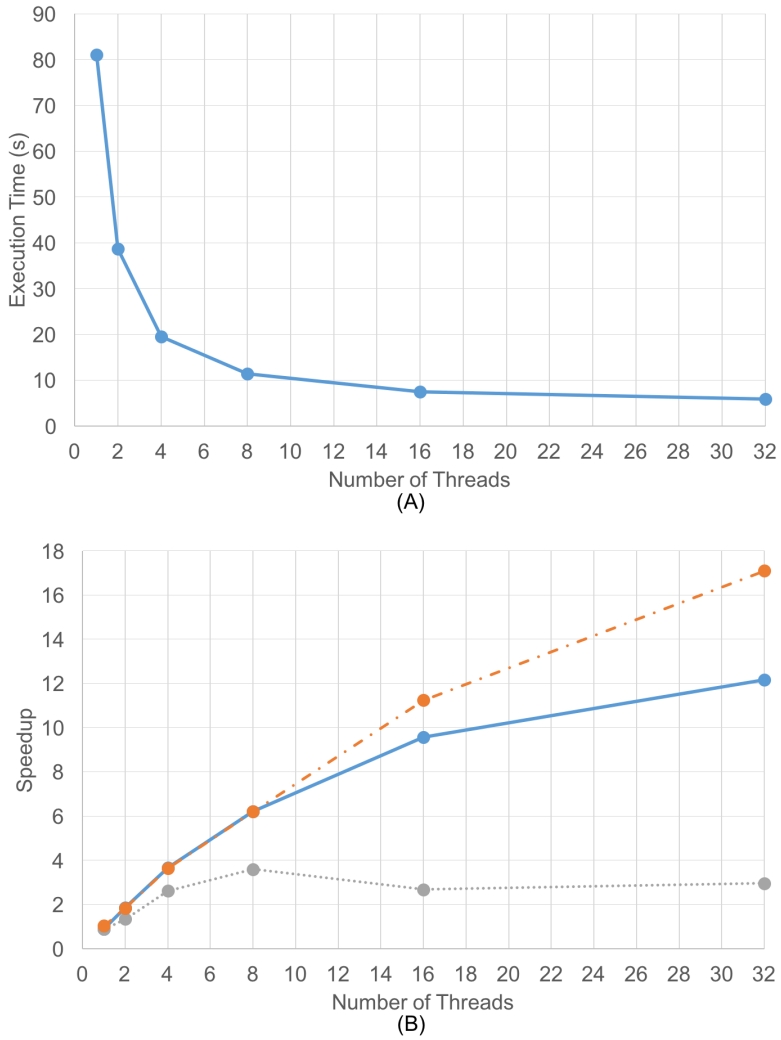


Figure 6.4: 3D VF ABM computation-only performance scalability. Graphs showing (A) execution time and (B) speedup of the 3D VF ABM as a function of number of threads. Notice that the average speedup of model function routines (orange dotted) is much higher than that of the update routines (grey dotted). This is because, as opposed to the update routines, the model function routines perform more computations than memory access operations. Even with some memory-bound functions, the overall speedup of the program (blue solid) is still satisfactory.

6.2.2.2 Visualization Only Performance

Since coarse-grain tasks (excluding updates) take about 4.7 s while fine-grain tasks on the GPUs only take 2.5 s, there is an idle period on the GPUs waiting for the CPUs to finish the coarse-grain tasks. Our goal is to make the visualization component fast enough so that the 2.2-second window gap would allow us to integrate visualization with computation on the GPUs without increasing the total execution time.

The visualization component includes cell migration, chemical diffusion, and tissue damage tracking. The most time consuming component is the chemical diffusion, which requires an access of 154 million points of data during each iteration. As discussed earlier, data sampling was used to improve the visualization performance. The plot in Fig. 6.5 shows the execution time and screenshots of chemical visualization using different sampling window widths. The visualization of the entire world looked almost identical for up to 6^3 sampling windows. Results showed that looking at the entire simulation area, enough information was retained by using 6^3 sampling. By using host-side constant sampling, the visualization runtime of chemical diffusion went down significantly from 23 s to 0.4 s.

There are circumstances in which the users may want to zoom in to see chemical distribution around the most active area, namely the wound area. For this, we can use adaptive sampling to provide higher resolution to the wound site, to ensure smooth images even when zooming in while keeping the performance optimal. Observed from the visualization view of the entire simulated volume using constant

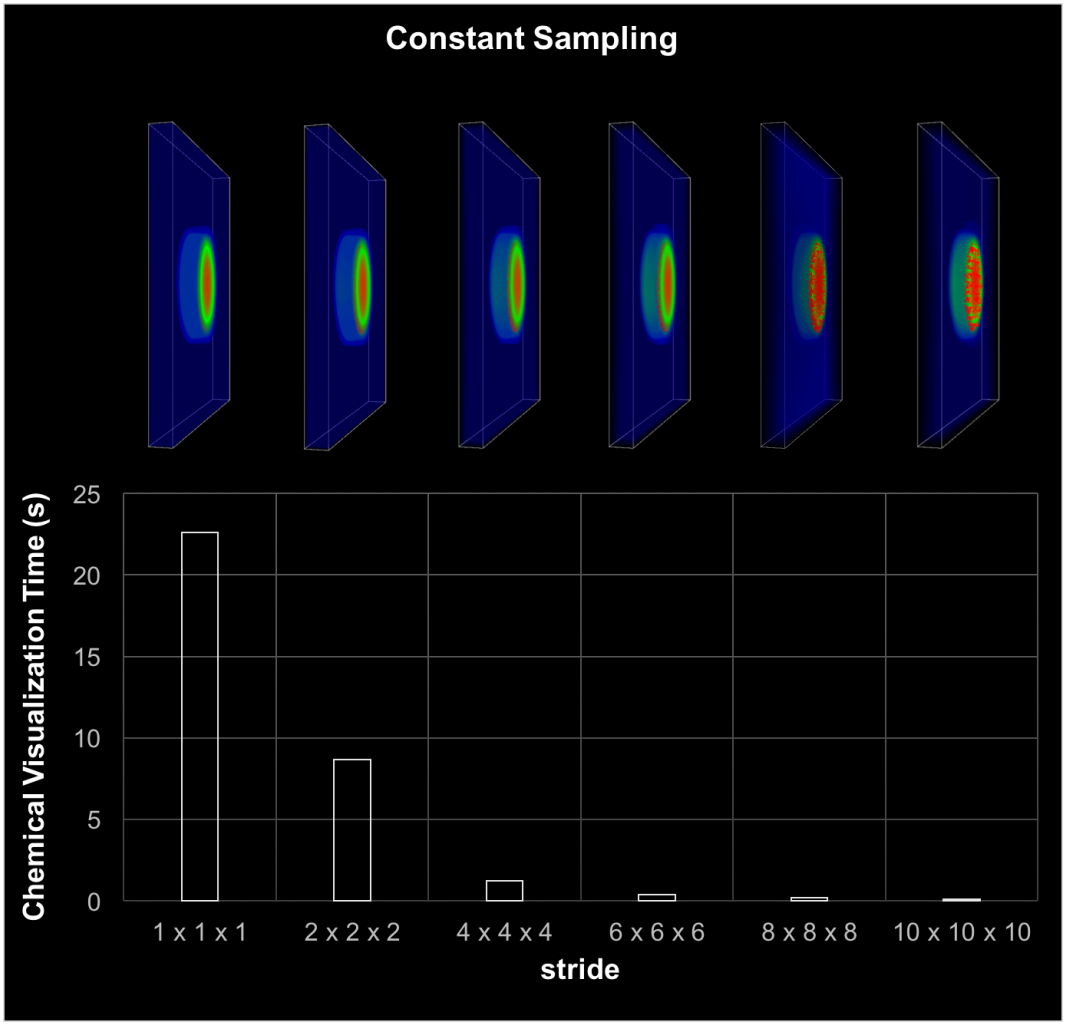


Figure 6.5: 3D VF ABM visualization-only performance. This chart shows visualization screenshots and corresponding execution time for different sampling resolution. The stride denotes the gap between two consecutive sampled points, thus the higher the stride the coarser the sampling. The visual appearance of the each sampling case looks almost identical for up to stride 6 or 6^3 sampling windows. Enough information can be retained by using 6^3 sampling.

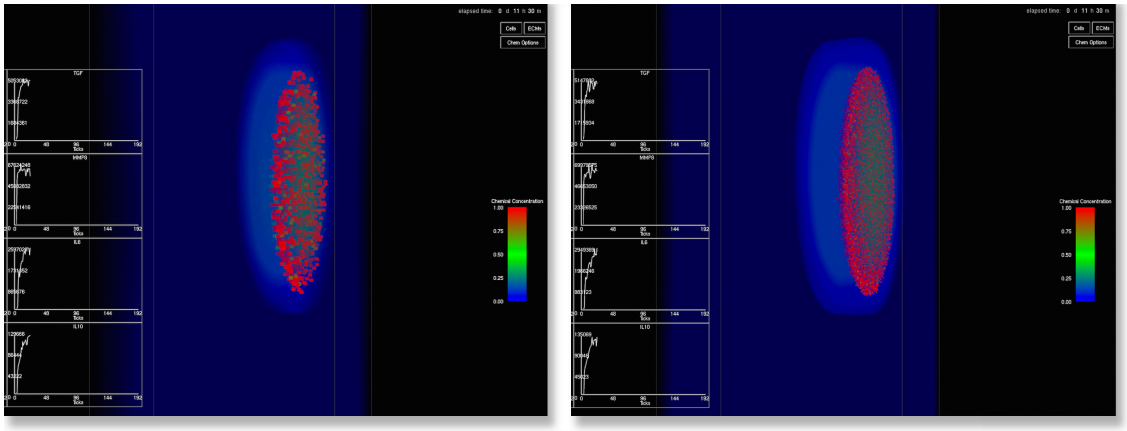


Figure 6.6: Screenshots comparison of 6^3 constant sampling windows (left) and $2-4-6$-adaptive sampling resolution (right) when zoomed in to high-activity area

sampling, the appearance of sampled simulation was indistinguishable from 1^3 up to 6^3 sampling. Thus, for optimal performance, the least important areas in adaptive sampling are set to use the coarsest resolution of 6^3 sampling windows. The medium and most important areas are then sampled with finer 4^3 and 2^3 windows, respectively. With $2-4-6$-sampling resolution, as shown in Fig. 6.6, the resolution of the visualization improved significantly, while the execution time of the visualization component only increased to 1.9 s (from 0.4 s), which is still below our visualization time budget, hence no increase in the overall execution time.

6.2.2.3 Computation-Visualization Performance

Since the visualization execution time was reduced from 23 s down to 0.4 s using data sampling for chemical diffusion, the visualization execution could then be placed in the idle period on one of the GPUs. By placing the visualization execution in a GPU idle gap, the total execution time remained unchanged at 6.2 s

per iteration on average. This fast execution time enabled the simulation to execute remote computation, remote visualization, remote transmission of the result frame, and frame rendering on the client's machine in under 7 s/frame. This performance, as far as we know, is the fastest known complex ABM simulation and visualization at a similar scale.

For benchmarking purposes, the 3D VF ABM was compared to our previous and other ABM works of similar nature (Fig 6.8). The M. Tuberculosis (MTb) ABM [75] was benchmarked on a system with an NVIDIA GeForce 8800M GTX GPU, while GeForce GTX Titan was used for FLAME GPU immune system ABM [167]. Despite the differences in underlying hardware, MTb ABM simulation is arguably one of the most suitable works for performance comparison with the 3D VF ABM. The 2D MTb ABM simulated a complex multi-scale biological system of agents that communicate via chemical signals, which aligned in most respects with the 3D VF ABM. The human immune system ABM was built on a widely used HPC ABM platform, FLAME GPU [167]. Although this ABM executed the immune system at a much smaller timescale, the cell communication method is similar to other ABMs included in this performance comparison, i.e. communication via chemical signals. The FLAME GPU immune system ABM thus served as a good performance reference.

The 3D VF ABM was simulated at a scale physiologically representative of a human vocal fold. Such scale was not feasible to be implemented on ABM free-ware NetLogo [89]. Furthermore, to our best knowledge, no similar scale had been reported in any other publication. For a common throughput unit, the simulation

performance was measured in terms of environment *space unit per millisecond*. The space units represent the smallest granularity of the ABM environment. Depending on the model, the space units can be patches [93], grid points [75] or immobile tissue cells [167]. These quantities determine the ABM environment size. Therefore, the number of space units are proportional to the amount of work required to simulate the ABM environment in each iteration. For this reason, *space unit per millisecond* serves a reasonable throughput measure. The 3D VF ABM is capable of processing 25K patches/ms, which is about 900x, 63x, 2.3x and 2.4x the throughputs of NetLogo, MTb, FLAME GPU immune system ABM and the 2D VF ABM, respectively. The comparison of the model scale, complexity and performances are in Table 6.6. Of note, FLAME GPU can process roughly 1.9x more mobile agents than 3D VF ABM per time unit. The primary reason was that the time step used in FLAME GPU immune system ABM are smaller than that of our model in orders of magnitudes. This time scale difference caused their agent rules to be much less complex. For example, FLAME GPU immune system ABM would take roughly 18 hours to complete a 5-day simulation while the 3D VF ABM only takes less than half an hour. In addition, the 3D VF ABM offered a much more rigorous data visualization in real-time at a scale of over 100 times more mobile agents than that of FLAME GPU immune system ABM.

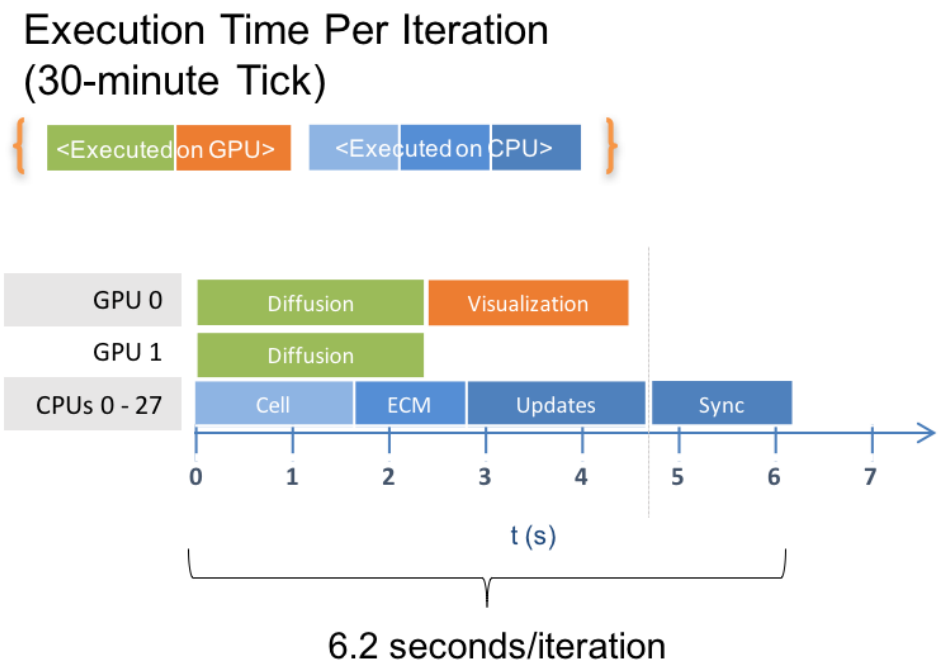


Figure 6.7: 3D VF ABM simulation suite performance. Chart demonstrating overlapped visualization and computation executions on GPUs and CPUs in terms of execution time.

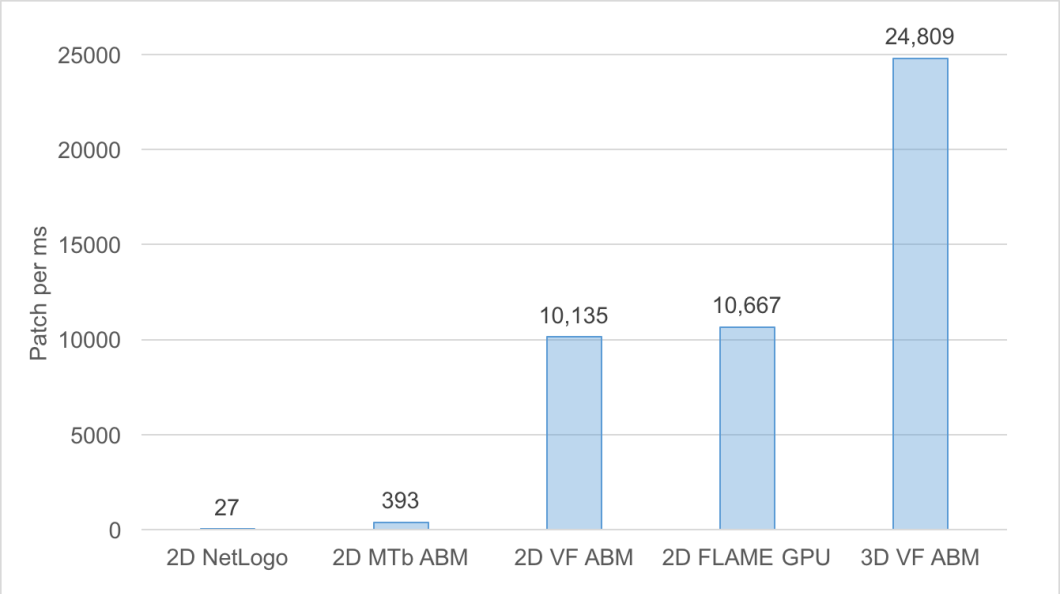


Figure 6.8: Processing power of 3D VF ABM vs. existing work comparison. This bar chart compares workload and execution time in terms of number of patches (i.e. lattice points, grid points, stationary cells) per ms between the 3D VF ABM to other bio-simulation ABM work. Notice that the 3D VF ABM is capable of processing 25K patches/ms, or about 900x, 63x, 2.3x and 2.4x more patch processing power than NetLogo, MTb ABM [75], FLAME GPU immune system ABM [167] and the earlier 2D VF ABM work [14].

	# Patches	# Agents	# Chemical Types	# ECM Protein Types	Time Step	Average Execution Time per Iteration (s)
2D MTb ABM	16.4 K	3.2 K	1	0	10 min	0.042
2D NetLogo VF ABM	1.0 M	114.0 K	8	3	30 min	36.6
2D FLAME GPU	320.0 K	160.1 K	1	0	0.2 s	0.03
2D VF ABM	1.9 M	228.0 K	8	3	30 min	0.19
3D VF ABM	153.8 M	16.9 M	8	3	30 min	6.2

Table 6.6: Performance and scale comparison of 3D VF ABM with existing high-performance ABM work of similar nature

6.2.3 3D High-Interactivity Human Model

6.2.3.1 Computation-Visualization Performance

As discussed in Section 6.2.2, the original frame rate of the human VF visualization was bounded by the compute time of approximately 7 s per iteration. This resulted in poor interactivity due to the 0.14 fps frame rate. With the scheduling technique described Section 3.3, the frame rate improved to 7.2 fps. By using the subvolume size recommended by the heuristic proposed in Section 5.3.1.2 for HADC optimization, the visualization performance improved significantly to 42.8 fps.

For performance evaluation purposes, the human VF ABM was compared against our previous and other similar ABM work (Table 6.7). The bacteria-macrophage-antibiotic ABM was implemented with FLAME GPU [167]. FLAME GPU is a widely used modern HPC ABM framework, and thus, serves as a good performance standard. Additionally, we included a well regarded high-performance

	#**Data Points** ($\times 10^6$)	***In Situ* Support**	**Frame Rate** (fps)	**Hardware**
MegaMol [190]	100	✓	10	Intel Core i7-2600 (16 GB RAM) NVIDIA GeForce GTX TITAN
2D FLAME GPU[a] [167]	0.48	×	33[b]	Intel Core i7 (8 GB RAM) NVIDIA 830M GeForce
3D VF ABM (original [16])	1200	✓	0.13	Intel Xeon E5-2699 v4 (128 GB RAM) NVIDIA Tesla M40
3D VF ABM (optimized)	1700	✓	42.8	Intel Xeon E5-2699 v4 (128 GB RAM) NVIDIA Tesla M40

[a]Bateria-Macrophage-Antibiotic ABM
[b]Derived frame rate

Table 6.7: Framerate and scale comparison of 3D VF ABM with existing biological visualization platforms

visualization prototype, MegaMol for comparison [190]. Despite MegaMol being an atomic-level visualization prototype, this powerful visualization tool is particle-based. Since particle-based simulation engine offer frameworks that can be adapted to visualize cellular-level data, MegaMol is well-suited to serve as our visualization performance comparison base. Our VF ABM is able to process data orders of magnitude larger than FLAME GPU at a similar frame rate. Furthermore, we are able to visualize 17x more data points than MegaMol at a 4.2x better frame rate. It is worth noting that MegaMol does perform more sophisticated rendering process than our framework. However, while our framework couples the visualization engine with grid-based real-time data generation, MagaMol did not support this feature [190].

6.3 Performance-Accuracy Trade-Off Studies

6.3.1 Device-Side Sampling

To understand the trade-offs between speed and accuracy as a result of sampling using GPU texture memory, an experiment using different sampling strides was performed and the results are shown in Fig. 6.9. The gray area of the plot in Fig. 6.9(a) represents the cases where chemical data memory requirement is too large for the GPUs. The performance jumps as we go from the gray area to the white since sufficient GPU memory means less data movements. The plot in Fig. 6.9(b) demonstrates the accuracy trade-offs in terms of normalized root-mean-squared error (NRMSE) that resulted from sampling. The white area of the plot in Fig. 6.9(b) represents the acceptable range of sampling strides, where the sampling results fit in the GPU memory and the average accuracy is higher than 95%. Fig. 6.9(c) compares the results of the visualization without sampling and with sampling ($6 \times 6 \times 6$ window). Up to a 6^3 window, the gradient details are preserved very well. However, increasing the sampling window size resulted in a noticeable loss of details visually and numerically (error rising above 5% in Fig. 6.9(b)). Thus, the 6^3 sampling window was used for the final performance evaluation. Fig. 6.10(a) shows a comparison of real rat vocal fold images that were obtained experimentally [210] to a model-generated image. The left most image in Fig. 6.10(a) that was obtained from an uninjured control rat vocal fold has lower collagen content (red), whereas the middle image from a scarred vocal fold shows a higher level of collagen content.

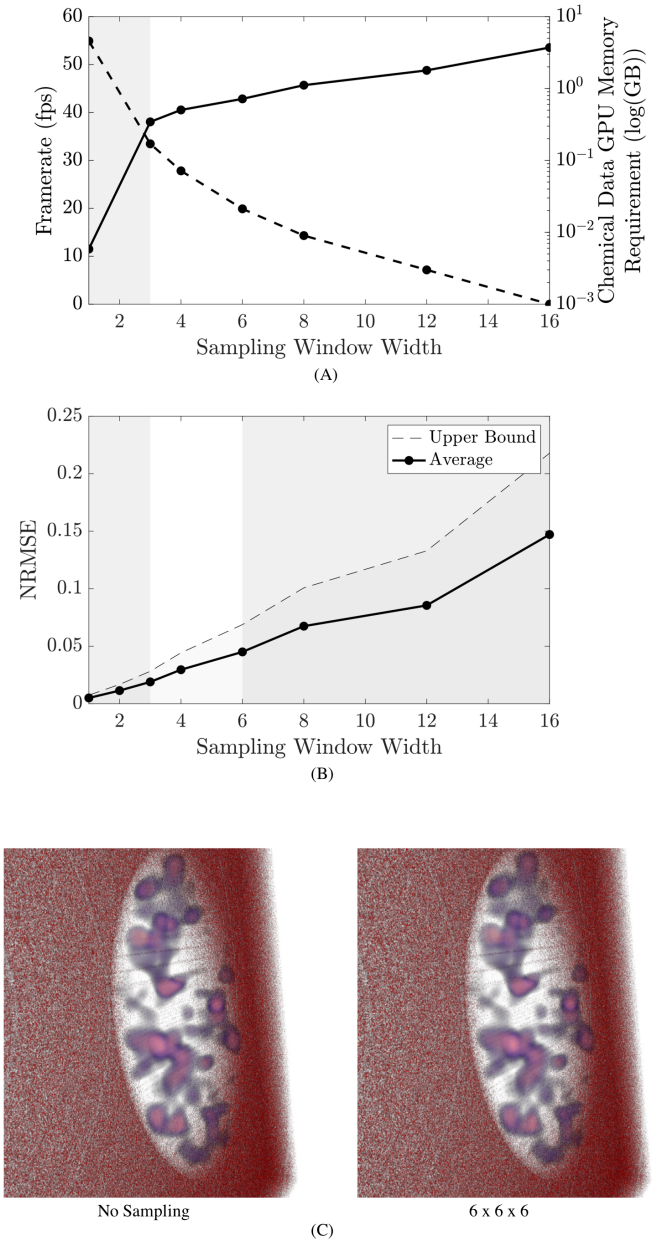
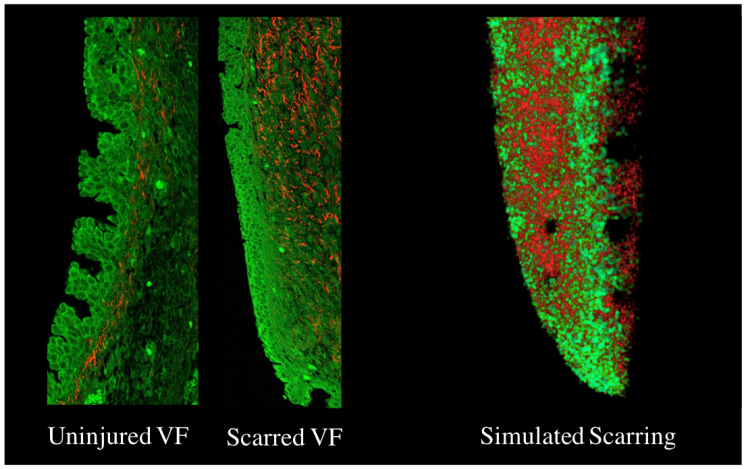
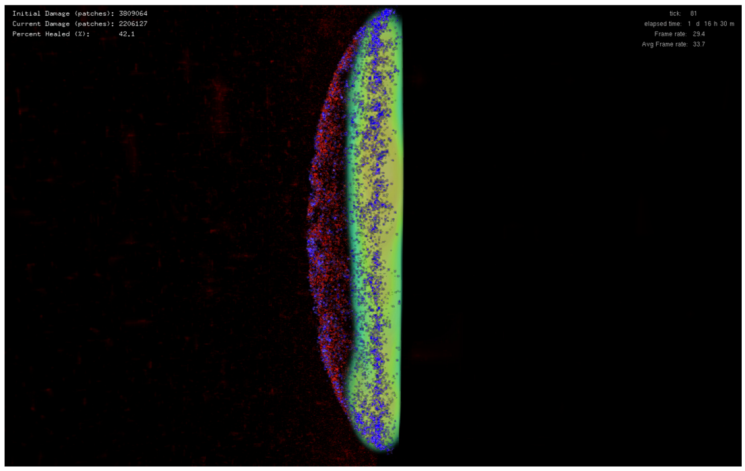


Figure 6.9: A study comparing the performance gain in (a) (frame rate and GPU memory requirement reduction with solid and dashed lines, respectively) to the normalized root-mean-squared error (RMSE) shown in (b) with respect to different sampling window widths. An example of a comparison of the visualization quality between the original data (*left*) and sampled data with a $6 \times 6 \times 6$ sampling window (*right*) is demonstrated in (c). Images of vocal fold tissue in (c) show an early stage with not much collagen (red) in the wound area (white oval) for a clear view of chemical (TNF in pink-purple) visualization.



(A)



(B)

Figure 6.10: (a) Comparison of vocal fold image (collagen in red, elastin and cells in green) of real rat vocal fold (uninjured control on the left and scarred in the middle) [210] and a zoomed image obtained from VF ABM simulation (right). (b) Visualization of human vocal fold both ECM proteins and signaling proteins (chemical gradients in turquoise-pear) during the healing process. The ECM proteins include collagen (red), hyaluronic acid (blue), and elastin (green). The healing and elapsed time stats are displayed in the top-left and top-right corner of the screen, respectively.

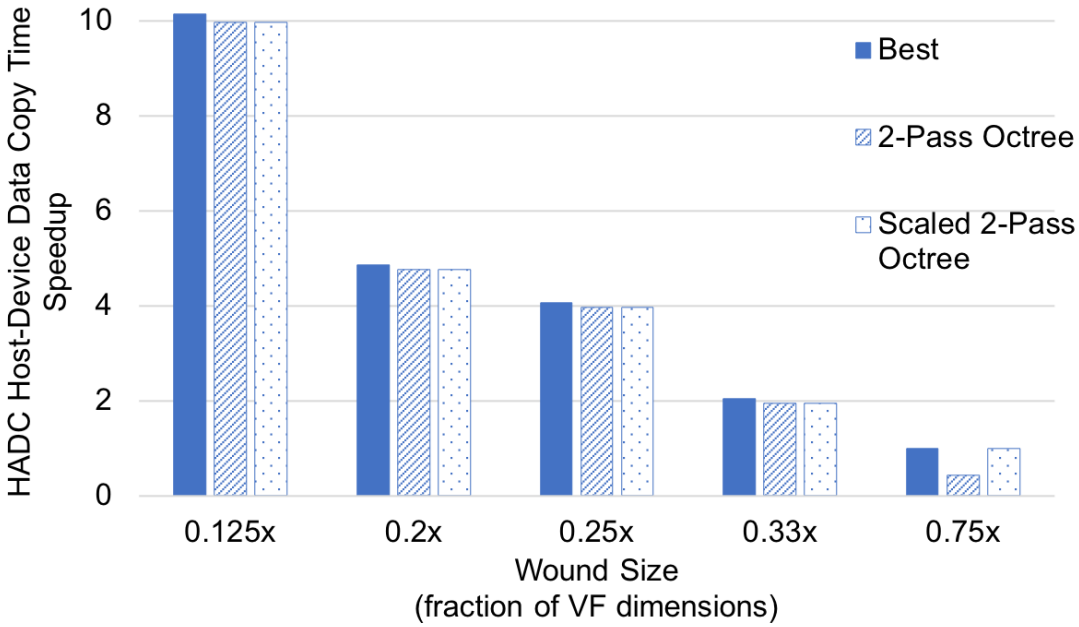


Figure 6.11: Data transfer time speedup using HADC. For each wound configuration, the model was run with different HADC partition sizes. The partition size that resulted in the best average data transfer time was used to compute the speedup (solid blue) and compared with speedup using partition size recommended by heuristic described in Section 5.3.1.2. The 2-pass octree technique (striped) worked well for small and medium size wounds. In contrary, the scaled 2-pass octree using the scaled recommendation from smaller wound model, m, for m' with larger wounds work well for all wound sizes (dotted).

The rightmost image obtained from a VF ABM simulation of an injured vocal fold shows a similar collagen content to the scarred vocal fold image. A screenshot of the *in situ* visualization of both ECM and signaling proteins, along with healing and timing stats is shown in Fig. 6.10(b). The visualization of ECM and signaling proteins in all model-generated images (Fig. 6.10) used HADC and 6^3 sampling window, respectively.

6.3.2 HADC

To evaluate the quality of the subvolume size recommendation made by the heuristic proposed in Section 5.3.1.2, an experiment was run with different HADC partition sizes for different model and wound configurations. The plot in Fig. 6.11 compares best speedup obtained from the experiments with speedup gained using 2-pass octree and scaled 2-passed octree, respectively. Compared to the best sub-volume sizes obtained experimentally (brute force), the 2-pass octree technique performed relatively well for small and medium size wounds (sizes up to $\frac{1}{3}$ of the VF size). However, it did not work well for large wound sizes as it recommended a sub-optimal partition. Thus, it is best for the modeler to use the discussed scaling heuristic to obtain the recommendation from smaller wound model, m and scale the partition size for m' with larger wounds.

6.4 Model Verification

The trends of the 3D rat and human VF ABM outputs were *qualitatively* verified using the pattern-oriented analytical approach [1, 211, 212]. The purpose of qualitative verification was to ensure that the dynamics of the model reflect what is expected in the wound healing literature and the available experimental data [211–214].

6.4.1 Human Model

Cell population and ECM protein trends were compared against known patterns reported in wound healing literature as summarized in Table 6.8 [215–224]. Fig. 6.12 shows cellular and molecular outputs of the VF ABM from a 7-day simulation. The model predicted a peak neutrophil population at the end of day 1 and significant decreases in day 2. The model also reproduced a peak of macrophage population around day 2 and a downward trend from the beginning of day 3 onward. Furthermore, the fibroblast proliferation started around the end of day one in the simulation. Trends of these specific cell populations agreed well with the known patterns in wound healing literature (Table 6.8). For ECM outputs, the VF ABM reproduced the trends of collagen but not of hyaluronan. In particular, both empirical and ABM results showed the accumulation of collagen starting from Day 3. The ABM predicted an earlier accumulation of hyaluronan (Day 1) compared to empirical data (Day 3). This early hyaluronan accumulation might be related to high levels of TNF-α, TGF-β, FGF and IL-1β that stimulated the secretion of hyaluronan by fibroblasts in the model. More data and calibration are needed for further investigation.

6.4.2 Rat Model

Due to the data availability, only a subset of chemicals was compared against the empirical data [213, 214]. This subset includes measured mRNA levels of three inflammatory mediators (TNF-α, TGF-β, and IL-1β) out of 8 that are simulated by

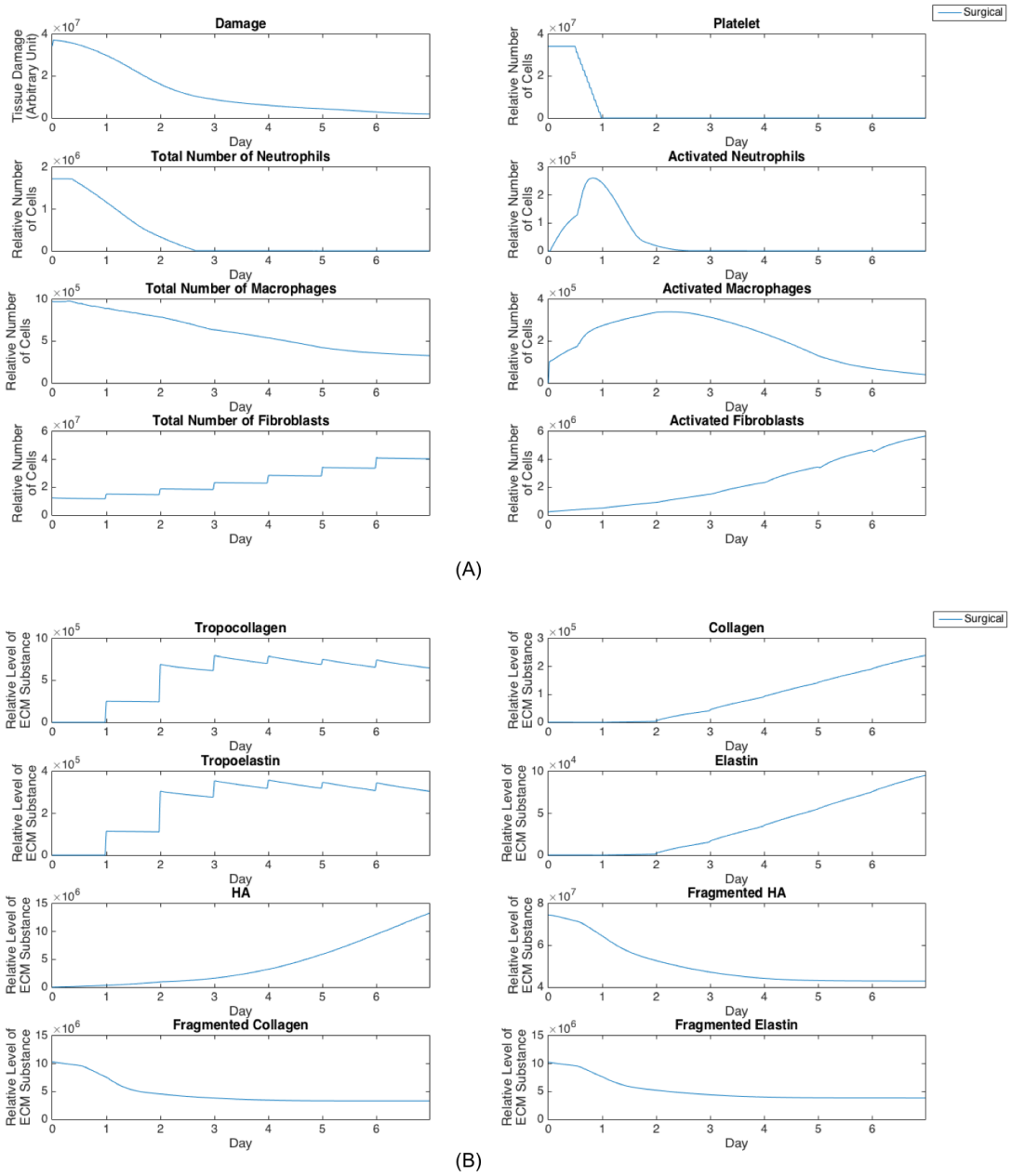


Figure 6.12: Simulation output of 3D human VF ABM for pattern-oriented model verification. (A) Damage magnitude, cells, chemicals and (B) ECMs trends.

Validation Patterns	Source References
Neutrophils arrive at wound site in first few hours	[218–221]
Neutrophil number is at maximum by day 1 or 2	[218–221]
Neutrophil number decreases rapidly around day 3 or 4	[218–221]
Macrophage number is at maximum by days 2 to 4	[218–221]
Fibroblasts start proliferation on day 1	[216]
Fibroblast number decreases significantly on day 7 and stays low until day 14	[216, 218–221]
Hyaluronan is first seen on day 3 and peaks at day 5, starts to drop significantly at day 7, and then remains at low level until day 14	[215, 217, 222, 224]
Peak of accumulated hyaluronan content occurs at same time as peak of inflammatory cells (neutrophils and macrophages)	[222, 223]
Hyaluronan level is generally lower than for uninjured vocal folds after injury throughout healing period	[215, 217]
Collagen type I curve is sigmoid-shaped	[220, 221]
Collagen type I is first seen on day 3 and peaks on day 5	[215, 217]
Collagen type I level is generally higher than for uninjured vocal folds after injury throughout healing period	[215, 217]

Table 6.8: Summary of patterns used for qualitative verification of 3D VF ABM [1]

the model. The comparison of the model outputs and the empirical data are shown in Fig 6.13. The ABM generated a peak of TNF-α after 13 hours (26 ticks) of injury, whereas this peak occurred at hour 8 (tick 16) in the empirical data. For IL-1β, the model generated a peak at hour 12 (tick 24), where the peak was observed at hour 8 in the empirical data. Overall, the ABM-predicted peaks for TNF-α and IL-1β lagged behind the experimentally observed peaks by 4-5 hours. The discrepancy between the model outputs and literature data may be explained as follows. First, since TNF-α and IL-1β were down-regulated by TGF-β and IL-10 via macrophages and fibroblasts, a possible reason for the peak delay could be an insufficient strength of TGF-β or IL-10. Second, since no empirical data were reported between hour 8 and 16, a peak between this interval might have been missed experimentally. More

empirical data are needed for further investigation. For TGF, the model missed predicting the spike at hour 1. However, the sub-linear upward trend from hour 4 till the end of the simulation predicted by the model matched with that of the empirical data. In sum, the 3D rat VF ABM trajectories of inflammatory mediators showed a few discrepancies when comparing with the empirical vocal fold data in literature. Despite these few discrepancies, the overall dynamics of the VF ABM outputs are consistent with those seen in the empirical data. Note that for this 3D VF ABM to be clinically ready, more experimental data is needed to calibrate the model.

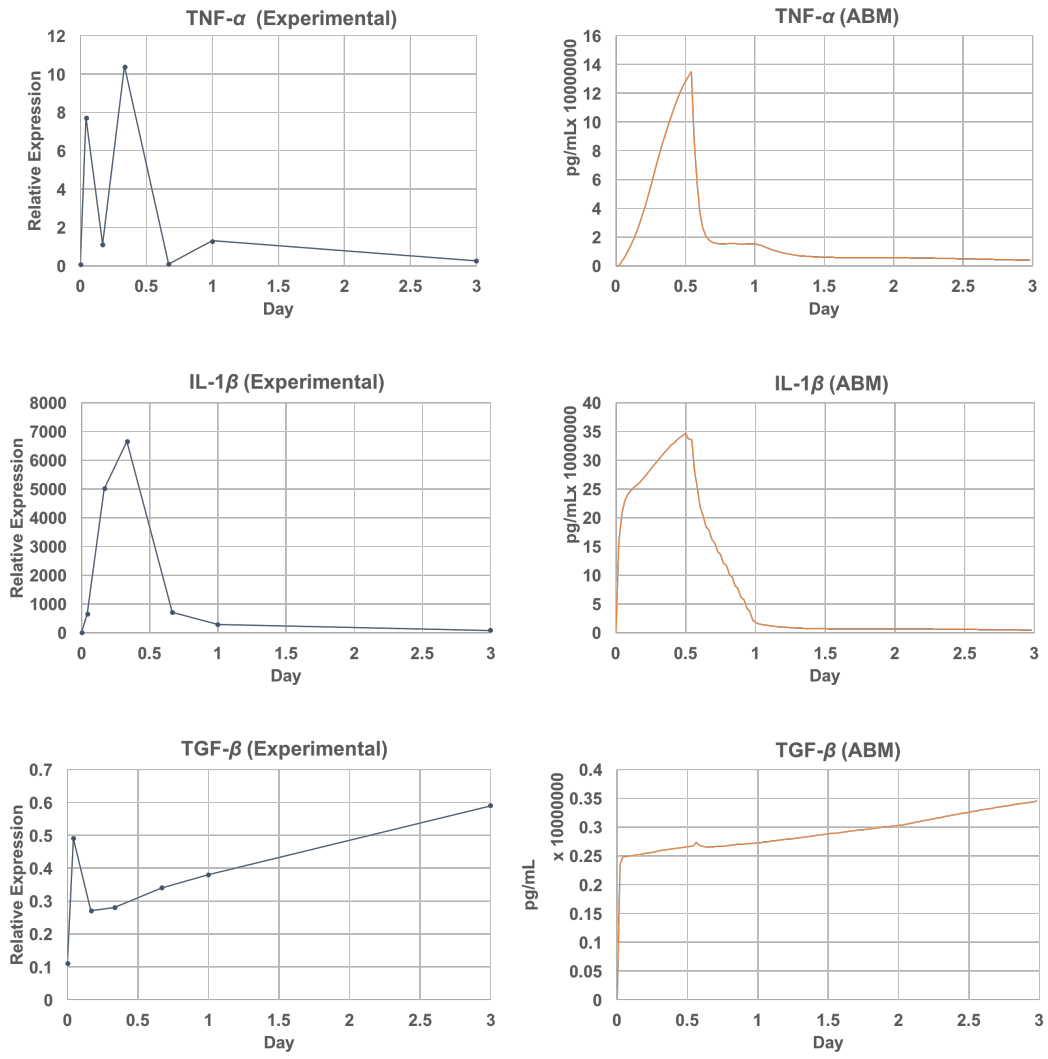


Figure 6.13: Empirical data vs. 3D rat VF ABM simulation output plot. Qualitative verification of the model output (left) against empirical data [213, 214] (right). The set of verified chemicals includes TNF-α, TGF-β, and IL-1β.

Chapter 7: Concluding Remarks and Future Perspectives

7.1 Conclusion

In this dissertation, high-performance computing (HPC) techniques for cellular-level inflammation and repair models were developed to optimize the simulation at both task-scheduling and task-execution levels. At the task-scheduling level, we started by building a task categorization method for wound healing applications that became the basis of our heterogeneous platform task mapping methodology. This task mapping method was then used in different variations of CPU-GPUs scheduling schemes to optimally utilize available computing resources on single-node host-accelerators machine configurations. At the task-execution level, tasks on CPU underwent further parallelization using OpenMP, while task-specific optimizations and communication reduction techniques were developed for GPU tasks. Both task-scheduling and task-execution level optimizations were applied first to 2D models, then more sophisticated optimized variations were applied to 3D models.

Since wound healing models are multi-scale in nature (consisting of molecular and cellular level operations), the finer scale processes of chemical diffusion were accelerated using FFT-based convolution. The computations of accelerated diffusion were then assigned to the GPU as these FFT computations involved large amounts

of data. The cellular-level operations such as cell seeding, cell proliferations and cell functions were mapped to multi-core CPU and sped up using OpenMP. This scheduling scheme was applied to 2D vocal fold inflammation and repair model consisting of over 200k biological cells and generated over 20 million data points. The model computation was 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. Performance of a 2D simulation of human vocal fold inflammation and wound healing showed 35x and 7x 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.

Next, we transitioned to 3D VF models. Since the 3D human model is substantially more computationally demanding, we extended the scheduling scheme to not only overlap fine-grain and coarse-grain computations, but also overlap the visualization operations. The visualization component was optimized with an adaptive sampling scheme to accelerate the rendering process while maintaining the precision of the displayed visual results. This visualization optimization ensured that the execution of the visualization operations is fast enough for the visualization cost to be completely hidden behind the CPU task execution. This extended scheduling scheme, along with adaptive sampling, resulted in effectively free visualization.

These techniques, thus, improved the performance of the whole simulation suite significantly, enabling the simulation and visualization of over 17 million biological cells and 1.2 billion chemical data points in under 7 seconds per iteration.

Finally, the support for GPU hyper-tasking (GHT) was integrated to our scheduling scheme. The benefits of GHT are two folds. First, the GPUs responsible mainly for fine-grain computations can hyper-task between fine-grain computations and ray-casting (visualization subtask). This allows these non-screen-attached GPUs to aid in visualizing their local data and minimize peer communications. Second, GHT allows the rendering GPU to lighten the computational loads of its peers by executing leftover fine-grain subtasks, while maintaining high level of visualization interactivity. Furthermore, the communications between the CPU host and the rendering GPU are minimized using an activity-aware data transfer technique to reduce redundancy in host-device data copies. The combination of GHT scheduling scheme and redundancy communication reduction allowed for an integration of additional 461 million extracellular matrix (ECM) protein data points (on top of 1.23 billion chemical data points) with an increase in visualization interactivity. Our simulation platform is capable of visualizing these 1.7 billion numerical data points, as they are being generated (i.e. *in-situ* visualization), with an average frame-rate of 42.8 fps. This results in real-time interactivity that allows users explore such large data sets on a remote server without any lag.

7.2 Generalization of HPC Techniques

The optimization discussed may seem to be tailored specifically for wound healing applications. However, chemical diffusion is a crucial part of most models of biological systems such as hormones in the endocrine system and pharmacokinetics of drug infusions. In addition, particle diffusion is seen in even larger number 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. Furthermore, the diffusion equation (Eqn. 4.1) 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 [225], Schrodinger equation for a free particle [226], thermal diffusivity and financial mathematics. And more importantly, if we generalize the scheduling schemes, the computational overlap techniques 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) Data-intensive GPU suitable task

(c) Visualization task (optional)

If all the three properties above hold, on of the CPU/GPUs computation overlap techniques can be applied to improve efficiency and speed up any system modeling implementation on heterogeneous (CPU/GPU) computing platforms. More specifically, the scheduling technique discussed in Section 3.2 can be applied to any application that exhibits all three of the aforementioned properties with low complexity or no visualization tasks. On the other hand, the techniques discussed in Section 3.3 and 3.4 can be used to improve the performance of applications that exhibit all required properties with high complexity and time consuming visualization tasks.

7.3 Future Directions

Currently, the cellular level tasks assigned to the CPU are the bottleneck of the simulation performance. The OpenMP parallelization developed in this work has likely reached the limit of the utilization of a single multi-core CPU. To further enhance the performance, distributed computing need to be considered. There are a number of existing ABM simulation frameworks that provide parallelization support on a cluster through MPI [227]. However, none of the existing distributed ABM frameworks integrated heterogeneous computing techniques to fully utilize both hosts and accelerators concurrently on all compute nodes. This leaves a window of opportunity to extend the heterogeneous computing techniques on a single node described in this dissertation, and provide heterogeneous distributed comput-

ing support through the combination of OpenMP, CUDA and MPI. In addition to the MPI support, an adaptation of the HADC technique is also worth explored. This activity-aware data transfer technique assumed constant subvolume size. An adaptive version that supports variable subvolume sizes could result in better performance. Since, the HADC technique was designed to minimize host-device communications on a single compute node, further improvements/adaptation have the potential to increase the efficiency of an MPI implementation by reducing node-node communications in a distributed computing environment.

The ABM implemented in this work assumed a regular grid simulation environment. However, this prevents the modelers from simulating arbitrary, possibly more realistic, organ shapes. The model implemented in this work used the regular grid environment to represent a flattened lamina propria section of a vocal fold, which is soft and pliable tissue. One drawback from this model is the inability to mimic curvature of the structure present in a real vocal fold, which could have suppressed the effects the shape has on events like cell migration. Possible ways to address this limitation is to use indirect addressing to reduce memory footprints of empty spaces for non-rectangular simulation volume [228], or spatial indices to represent more realistic 3D volume and efficiently resolve neighbor partitions [229].

At present, cell visualization is supported in our 2D and 3D versions, but not the high-interactivity 3D version of VF ABM. To integrate cell visualization in the high-interactivity 3D version, the runtime of cell visualization can be improved through streaming of cell information (types and locations) from the host to device. Since the number of cells present in the simulation changes from one iteration

to another, determining the optimal size of the device-side cell information buffer can be challenging. A work exploring this idea would, thus, be beneficial to most large-scale multi-agent applications. Lastly, a systematic way of verifying the visualization outputs through comparison with empirically obtained images can provide substantial assistance to researchers in testing and refining their models.

Bibliography

[1] Nicole YK Li, Yoram Vodovotz, Patricia A Hebda, and Katherine Verdolini Abbott. Biosimulation of inflammation and healing in surgically injured vocal folds. *The Annals of otology, rhinology, and laryngology*, 119(6):412, 2010.

[2] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[3] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[4] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openaccfirst experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[5] Ha-Nguyen Tran and Erik Cambria. A survey of graph processing on graphics processing units. *The Journal of Supercomputing*, 74(5):2086–2115, 2018.

[6] Yin-Te Tsai. An overview of machine learning and hpc in open sources for bioinformatics. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 1338–1342. IEEE, 2018.

[7] Alberto Cano. A survey on graphic processing unit computing for large-scale data mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(1):e1232, 2018.

[8] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-of-core gpu memory management for mapreduce-based large-scale graph processing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–229. IEEE, 2014.

[9] Jing Wu and Joseph JaJa. Optimized fft computations on heterogeneous platforms with application to the poisson equation. *Journal of Parallel and Distributed Computing*, 74(8):2745–2756, 2014.

[10] Stijn Heldens, Ana Lucia Varbanescu, and Alexandru Iosup. Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pages 62–65. IEEE, 2016.

[11] K Abhishek, N Sreenivasa, and S Balaji. Accelerated simulation of cell biological systems using heterogeneous parallel processing platforms-a survey. In *International Conference for Phoenixes on Emerging Current Trends in Engineering and Management (PECTEAM 2018)*. Atlantis Press, 2018.

[12] Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of simulation*, 4(3):151–162, 2010.

[13] Tomaž Velnar, Tracey Bailey, and Vladimir Smrkolj. The wound healing process: an overview of the cellular and molecular mechanisms. *Journal of International Medical Research*, 37(5):1528–1542, 2009.

[14] Nuttiiya Seekhao, Caroline Shung, Joseph JaJa, Luc Mongeau, and Nicole YK Li-Jessen. Real-time agent-based modeling simulation with in-situ visualization of complex biological systems a case study on vocal fold inflammation and healing. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[15] Nuttiiya Seekhao, Joseph JaJa, Luc Mongeau, and Nicole YK Li-Jessen. In situ visualization for 3d agent-based vocal fold inflammation and repair simulation. *Supercomputing frontiers and innovations*, 4(3):68, 2017.

[16] Nuttiiya Seekhao, Caroline Shung, Joseph JaJa, Luc Mongeau, and Nicole Yee Key Li-Jessen. High-performance agent-based modeling applied to vocal fold inflammation and repair. *Frontiers in Physiology (IMPACT FACTOR: 4.13)*, 9:304, 2018.

[17] Nuttiiya Seekhao, Grace Yu, Samson Yuen, Joseph JaJa, Luc Mongeau, and Nicole Yee Key Li-Jessen. High-performance host-device scheduling and data-transfer minimization techniques for visualization of 3d agent-based wound healing applications. *Accepted to the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2019.

[18] Christopher G. Willard, Addison, Laura Segervall, and Michael Feldman. Intersect360 Research hpc user site census: Processors. `http://www.intersect360.com/industry/reports.php?id=129`, 2017. Accessed: 2017-12-24.

[19] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[20] Lance Hammond, Basem A Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, (9):79–85, 1997.

[21] Balaji Venu. Multi-core processors-an overview. *arXiv preprint arXiv:1110.3535*, 2011.

[22] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[23] Abinash Roy, Jingye Xu, and Masud H Chowdhury. Multi-core processors: A new way forward and challenges. In *Microelectronics, 2008. ICM 2008. International Conference on*, pages 454–457. IEEE, 2008.

[24] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72. ACM, 2006.

[25] Intel. 2nd generation intel xeon scalable processors. `https://ark.intel.com/content/www/us/en/ark/products/series/192283/2nd-generation-intel-xeon-scalable-processors.html`. Accessed: 2019-04-27.

[26] W Hwu Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[27] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[28] David Luebke. Gpu architecture: Implications & trends. *SIGGRAPH 2008: Beyond Programmable Shading Course Materials*, 2008.

[29] NVIDIA. GeForce 256 the world's first gpu. `http://www.nvidia.com/page/geforce256.html`. Accessed: 2017-12-24.

[30] Thomas Scott Crow. Evolution of the graphical processing unit. *A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science, University of Nevada, Reno*, 2004.

[31] PCPartPicker. Price Trends gpus. `https://pcpartpicker.com/trends/price/video-card/`, 2017. Accessed: 2017-12-24.

[32] STEAM. Steam hardware & software survey: November 2017. `http://store.steampowered.com/hwsurvey/videocard/`, 2017. Accessed: 2017-12-24.

[33] NVIDIA. GEFORCE GTX 1060 10: Gaming perfected. `https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1060/`, 2017. Accessed: 2017-12-24.

[34] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[35] Qiang Wu, Yajun Ha, Akash Kumar, Shaobo Luo, Ang Li, and Shihab Mohamed. A heterogeneous platform with gpu and fpga for power efficient high performance computing. In *Integrated Circuits (ISIC), 2014 14th International Symposium on*, pages 220–223. IEEE, 2014.

[36] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[37] Top500 List. The top 500 list. `https://www.top500.org/lists/2018/11/`, 2018.

[38] Felix Friedrich, Oleksii Morozov, and Patrick Hunziker. A compute model for generating high performance computing socs on hybrid systems with fpgas. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers; Proceedings of*, pages 1–12. VDE, 2016.

[39] MV Ganeswara Rao, P Rajesh Kumar, and A Mallikarjuna Prasad. Implementation of real time image processing system with fpga and dsp. In *Microelectronics, Computing and Communications (MicroCom), 2016 International Conference on*, pages 1–4. IEEE, 2016.

[40] Joseph Marshall, Dale Rickard, Danielle Sova, Hubert Miller, Robert Lapihuska, Alan Dennis, and Michael Graziano. Heterogeneous high performance computing modules for next generation onboard processing. In *Aerospace Conference, 2017 IEEE*, pages 1–10. IEEE, 2017.

[41] Top500 List. The top 500 list statistics. `https://www.top500.org/statistics/list/`. Accessed: 2019-03-31.

[42] Addison Snell and Laura Segervall. Intersect360 Research hpc application support for gpu computing. `https://www.nvidia.com/content/intersect-360-HPC-application-support.pdf`. Accessed: 2019-03-31.

[43] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. *White Paper, Red Hat Inc*, 2003.

[44] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[45] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[46] William Gropp. Mpi 3 and beyond: why mpi is successful and what challenges it faces. *Recent Advances in the Message Passing Interface*, pages 1–9, 2012.

[47] Mpi forum. `http://www.mpi-forum.org`.

[48] Guy E Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[49] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[50] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[51] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[52] George Almasi. Pgas (partitioned global address space) languages. In *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011.

[53] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[54] Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[55] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[56] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[57] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. In *BMC bioinformatics*, volume 11, page S1. BioMed Central, 2010.

[58] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[59] James G Shanahan and Laing Dai. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 2323–2324. ACM, 2015.

[60] Charles M Macal. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, 10(2):144–156, 2016.

[61] Gary An, Qi Mi, Joyeeta Dutta-Moscato, and Yoram Vodovotz. Agent-based models in translational systems biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 1(2):159–171, 2009.

[62] Ferdi L Hellweger, Robert J Clegg, James R Clark, Caroline M Plugge, and Jan-Ulrich Kreft. Advancing microbial sciences by individual-based modelling. *Nature Reviews Microbiology*, 14(7):461, 2016.

[63] Thomas E Gorochowski. Agent-based modelling in synthetic biology. *Essays in biochemistry*, 60(4):325–336, 2016.

[64] Leigh Tesfatsion. Agent-based computational economics: Growing economies from the bottom up. *Artificial life*, 8(1):55–82, 2002.

[65] Leigh Tesfatsion. Agent-based computational economics: modeling economies as complex adaptive systems. *Information Sciences*, 149(4):262–268, 2003.

[66] Leigh Tesfatsion and Kenneth L Judd. *Handbook of computational economics: agent-based computational economics*, volume 2. Elsevier, 2006.

[67] Leigh Tesfatsion. Agent-based computational economics: A constructive approach to economic theory. *Handbook of computational economics*, 2:831–880, 2006.

[68] W Brian Arthur. Out-of-equilibrium economics and agent-based modeling. *Handbook of computational economics*, 2:1551–1564, 2006.

[69] Alessandro Caiani, Alberto Russo, Antonio Palestrini, and Mauro Gallegati. *Economics with Heterogeneous Interacting Agents: A Practical Guide to Agent-Based Modeling*. Springer, 2016.

[70] Adam J McLane, Christina Semeniuk, Gregory J McDermid, and Danielle J Marceau. The role of agent-based models in wildlife ecology and management. *Ecological Modelling*, 222(8):1544–1556, 2011.

[71] Robin Gras, Didier Devaurs, Adrianna Wozniak, and Adam Aspinall. An individual-based evolving predator-prey ecosystem simulation using a fuzzy cognitive map as the behavior model. *Artificial life*, 15(4):423–463, 2009.

[72] Adam Lampert and Alan Hastings. Stability and distribution of predator–prey systems: local and regional mechanisms and patterns. *Ecology letters*, 19(3):279–288, 2016.

[73] Adam J McLane, Christina Semeniuk, Gregory J McDermid, Diana F Tomback, Teresa Lorenz, and Danielle Marceau. Energetic behavioural-strategy prioritization of clarks nutcrackers in whitebark pine communities: An agent-based modeling approach. *Ecological Modelling*, 354:123–139, 2017.

[74] NY Li, Katherine Verdolini, Gilles Clermont, Qi Mi, Elaine N Rubinstein, Patricia A Hebda, and Yoram Vodovotz. A patient-specific in silico model of inflammation and healing tested in acute vocal fold injury. *PloS one*, 3(7):e2789, 2008.

[75] Roshan M D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 21. Society for Computer Simulation International, 2009.

[76] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[77] Nicole YK Li, Yoram Vodovotz, Kevin H Kim, Qi Mi, Patricia A Hebda, and Katherine Verdolini Abbott. Biosimulation of acute phonotrauma: an extended model. *The Laryngoscope*, 121(11):2418–2428, 2011.

[78] Bryan N Brown, Ian M Price, Franklin R Toapanta, Dilhari R DeAlmeida, Clayton A Wiley, Ted M Ross, Tim D Oury, and Yoram Vodovotz. An agent-based model of inflammation and fibrosis following particulate exposure in the lung. *Mathematical biosciences*, 231(2):186–196, 2011.

[79] Nicholas A Cilfone, Denise E Kirschner, and Jennifer J Linderman. Strategies for efficient numerical implementation of hybrid multi-scale agent-based models to describe biological systems. *Cellular and Molecular Bioengineering*, 8(1):119–136, 2014.

[80] Kyle S Martin, Silvia S Blemker, and Shayn M Peirce. Agent-based computational model investigates muscle-specific responses to disuse-induced atrophy. *Journal of Applied Physiology*, 118(10):1299–1309, 2015.

[81] Andrey Krekhov, Jürgen Grüninger, Rainer Schlönvoigt, and Jens Krüger. Towards in situ visualization of extreme-scale, agent-based, worldwide disease-spreading simulations. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, page 7. ACM, 2015.

[82] Zhihui Wang, Joseph D Butner, Romica Kerketta, Vittorio Cristini, and Thomas S Deisboeck. Simulating cancer growth with multiscale agent-based modeling. In *Seminars in cancer biology*, volume 30, pages 70–78. Elsevier, 2015.

[83] Zhenzhen Shi, Stephen K Chapes, David Ben-Arieh, and Chih-Hang Wu. An agent-based model of a hepatic inflammatory response to salmonella: A computational study under a large set of experimental data. *PloS one*, 11(8):e0161131, 2016.

[84] Marcel Ausloos, Herbert Dawid, and Ugo Merlone. Spatial interactions in agent-based modeling. In *Complexity and Geographical Economics*, pages 353–377. Springer, 2015.

[85] Stephanie S Godfrey, C Michael Bull, Richard James, and Kris Murray. Network structure and parasite transmission in a group living lizard, the gidgee skink, egernia stokesii. *Behavioral Ecology and Sociobiology*, 63(7):1045–1056, 2009.

[86] David J Crandall, Lars Backstrom, Dan Cosley, Siddharth Suri, Daniel Huttenlocher, and Jon Kleinberg. Inferring social ties from geographic coincidences. *Proceedings of the National Academy of Sciences*, 107(52):22436–22441, 2010.

[87] Thomas O Richardson and Thomas E Gorochowski. Beyond contact-based transmission networks: the role of spatial coincidence. *Journal of The Royal Society Interface*, 12(111):20150705, 2015.

[88] Thomas E Gorochowski and Thomas O Richardson. How behaviour and the environment influence transmission in mobile groups. In *Temporal Network Epidemiology*, pages 17–42. Springer, 2017.

[89] Uri Wilensky and I Evanston. Netlogo: Center for connected learning and computer-based modeling. *Northwestern University, Evanston, IL*, pages 49–52, 1999.

[90] P Van Liedekerke, A Buttenschön, and D Drasdo. Off-lattice agent-based models for cell and tumor growth: Numerical methods, implementation, and applications. In *Numerical Methods and Advanced Simulation in Biomechanics and Biological Processes*, pages 245–267. Elsevier, 2018.

[91] Nitish Chooramun, Peter J Lawrence, and Edwin R Galea. An agent based evacuation model utilising hybrid space discretisation. *Safety science*, 50(8):1685–1694, 2012.

[92] D Drasdo, A Buttenschön, and P Van Liedekerke. Agent-based lattice models of multicellular systems: Numerical methods, implementation, and applications. In *Numerical Methods and Advanced Simulation in Biomechanics and Biological Processes*, pages 223–238. Elsevier, 2018.

[93] U Wilensky. Netlogo dictionary. *NetLogo User Manual*, 3, 2015.

[94] Neil Bhattacharyya. The prevalence of voice problems among adults in the united states. *The Laryngoscope*, 124(10):2359–2362, 2014.

[95] Nelson Roy, Ray M Merrill, Susan Thibeault, Steven D Gray, and Elaine M Smith. Voice disorders in teachers and the general population effects on work performance, attendance, and future career choices. *Journal of Speech, Language, and Hearing Research*, 47(3):542–551, 2004.

[96] Ingo R Titze, Julie Lemke, and Doug Montequin. Populations in the us workforce who rely on voice as a primary tool of trade: a preliminary report. *Journal of Voice*, 11(3):254–259, 1997.

[97] Erkki Vilkman. Voice problems at work: a challenge for occupational safety and health arrangement. *Folia phoniatrica et logopaedica*, 52(1-3):120–125, 2000.

[98] Katherine Verdolini and Lorraine O Ramig. Review: occupational risks for voice problems. *Logopedics Phoniatrics Vocology*, 26(1):37–46, 2001.

[99] Katherine Jones, Jason Sigmon, Lynette Hock, Eric Nelson, Marsha Sullivan, and Frederic Ogren. Prevalence and risk factors for voice problems among telemarketers. *Archives of Otolaryngology–Head & Neck Surgery*, 128(5):571–577, 2002.

[100] D Fellman and S Simberg. Prevalence and risk factors for voice problems among soccer coaches. *Journal of Voice*, 31(1):121–e9, 2017.

[101] Lady Catherine Cantor Cutiva, Ineke Vogel, and Alex Burdorf. Voice disorders in teachers and their associations with work-related factors: a systematic review. *Journal of Communication Disorders*, 46(2):143–155, 2013.

[102] Lučka Boltežar and Maja Šereg Bahar. Voice disorders in occupations with vocal load in slovenia. *Slovenian Journal of Public Health*, 53(4):304–310, 2014.

[103] Regina Helena Garcia Martins, Henrique Abrantes do Amaral, Elaine Lara Mendes Tavares, Maira Garcia Martins, Tatiana Maria Gonçalves, and Norimar Hernandes Dias. Voice disorders: etiology and diagnosis. *Journal of Voice*, 2015.

[104] Steven Gray and Ingo Titze. Histologic investigation of hyperphonated canine vocal cords. *Annals of Otology, Rhinology & Laryngology*, 97(4):381–388, 1988.

[105] Michael M Johns. Update on the etiology, diagnosis, and treatment of vocal fold nodules, polyps, and cysts. *Current opinion in otolaryngology & head and neck surgery*, 11(6):456–461, 2003.

[106] Heather E Gunter. Modeling mechanical stresses as a factor in the etiology of benign vocal fold lesions. *Journal of biomechanics*, 37(7):1119–1124, 2004.

[107] Chao Tao and Jack J Jiang. Mechanical stress during phonation in a self-oscillating finite-element vocal fold model. *Journal of biomechanics*, 40(10):2191–2198, 2007.

[108] Nicole YK Li, Hossein K Heris, and Luc Mongeau. Current understanding and future directions for vocal fold mechanobiology. *Journal of cytology & molecular biology*, 1(1):001, 2013.

[109] Tsuyoshi Kojima, Mark Van Deusen, W Gray Jerome, C Gaelyn Garrett, M Preeti Sivasankar, Carolyn K Novaleski, and Bernard Rousseau. Quantification of acute vocal fold epithelial surface damage with increasing time and magnitude doses of vibration exposure. *PloS one*, 9(3):e91615, 2014.

[110] Stephanie Misono, Schelomo Marmor, Nelson Roy, Ted Mau, and Seth M Cohen. Multi-institutional study of voice disorders and voice therapy referral: report from the cheer network. *Otolaryngology–Head and Neck Surgery*, 155(1):33–41, 2016.

[111] Jennifer K Hansen and Susan L Thibeault. Current understanding and review of the literature: vocal fold scarring. *Journal of Voice*, 20(1):110–120, 2006.

[112] Gerhard Friedrich, Marc Remacle, Martin Birchall, Jean Paul Marie, and Christoph Arens. Defining phonosurgery: a proposal for classification and nomenclature by the phonosurgery committee of the european laryngological society (els). *European Archives of Oto-Rhino-Laryngology*, 264(10):1191–1200, 2007.

[113] Diane M Bless and Nathan V Welham. Characterization of vocal fold scar formation, prophylaxis and treatment using animal models. *Current opinion in otolaryngology & head and neck surgery*, 18(6):481, 2010.

[114] Shigeru Hirano, Masanobu Mizuta, Mami Kaneko, Ichiro Tateya, Shin-Ichi Kanemaru, and Juichi Ito. Regenerative phonosurgical treatments for vocal fold scar and sulcus with basic fibroblast growth factor. *The Laryngoscope*, 123(11):2749–2755, 2013.

[115] John W Ingle, Leah B Helou, Nicole YK Li, Patricia A Hebda, Clark A Rosen, and Katherine V Abbott. Role of steroids in acute phonotrauma: a basic science investigation. *The Laryngoscope*, 124(4):921–927, 2014.

[116] Jaime E Moore, Paul J Rathouz, Jeffrey A Havlena, Qianqian Zhao, Seth H Dailey, Maureen A Smith, Caprice C Greenberg, and Nathan V Welham. Practice variations in voice treatment selection following vocal fold mucosal resection. *The Laryngoscope*, 126(11):2505–2512, 2016.

[117] Edwin ML Yiu, Karen MK Chan, Nicole YK Li, Raymond Tsang, Katherine Verdolini Abbott, Elaine Kwong, Estella PM Ma, Fred W Tse, and Zhixiu Lin. Wound-healing effect of acupuncture for treating phonotraumatic vocal pathologies: A cytokine study. *The Laryngoscope*, 126(1):E18–E22, 2016.

[118] Nelson Roy. Optimal dose–response relationships in voice therapy. *International Journal of Speech-Language Pathology*, 14(5):419–423, 2012.

[119] Elise Baker. Optimal intervention intensity in speech-language pathology: Discoveries, challenges, and unchartered territories. *International Journal of Speech-Language Pathology*, 14(5):478–485, 2012.

[120] Katherine Verdolini Abbott, Nicole YK Li, Ryan C Branski, Clark A Rosen, Elizabeth Grillo, Kimberly Steinhauer, and Patricia A Hebda. Vocal exercise may attenuate acute vocal fold inflammation. *Journal of Voice*, 26(6):814–e1, 2012.

[121] Nicole YK Li, Fei Chen, Frederik G Dikkers, and Susan L Thibeault. Dose-dependent effect of mitomycin c on human vocal fold fibroblasts. *Head & neck*, 36(3):401–410, 2014.

[122] Mary Pannbacker. Treatment of vocal nodulesoptions and outcomes. *American Journal of Speech-Language Pathology*, 8(3):209–217, 1999.

[123] Steven M Zeitels, Roy R Casiano, Glendon M Gardner, Norman D Hogikyan, James A Koufman, and Clark A Rosen. Management of common voice problems: committee report. *OtolaryngologyHead and Neck Surgery*, 126(4):333–348, 2002.

[124] Kenneth MacKenzie, Audrey Millar, Janet A Wilson, Cameron Sellars, and Ian J Deary. Is voice therapy an effective treatment for dysphonia? a randomised controlled trial. *Bmj*, 323(7314):658, 2001.

[125] Hideki Nakagawa, Makoto Miyamoto, Toshiyuki Kusuyama, Yuko Mori, and Hiroyuki Fukuda. Resolution of vocal fold polyps with conservative treatment. *Journal of Voice*, 26(3):e107–e110, 2012.

[126] Chi-Te Wang, Li-Jen Liao, Mei-Shu Lai, and Po-Wen Cheng. Comparison of benign lesion regression following vocal fold steroid injection and vocal hygiene education. *The Laryngoscope*, 124(2):510–515, 2014.

[127] Daniela de Vasconcelos, Adriana Oliveira de Camargo Gomes, and Cláudia Marina Tavares de Araújo. Effectiveness of speech therapy in the treatment of vocal fold polyps. *Revista CEFAC*, 17(6):2009–2017, 2015.

[128] Nicole YK Li, Katherine Verdolini Abbott, Clark Rosen, Gary An, Patricia A Hebda, and Yoram Vodovotz. Translational systems biology and voice pathophysiology. *The Laryngoscope*, 120(3):511–515, 2010.

[129] Yoram Vodovotz, Marie Csete, John Bartels, Steven Chang, and Gary An. Translational systems biology of inflammation. *PLoS Comput Biol*, 4(4):e1000014, 2008.

[130] Amir K Miri, Nicole YK Li, Reza Avazmohammadi, Susan L Thibeault, Rosaire Mongrain, and Luc Mongeau. Study of extracellular matrix in vocal fold biomechanics using a two-phase model. *Biomechanics and modeling in mechanobiology*, 14(1):49–57, 2015.

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

[132] Kevin Burrage, Lindsay Hood, and Mark A Ragan. Advanced computing for systems biology. *Briefings in bioinformatics*, 7(4):390–398, 2006.

[133] Gary S Ayton, Will G Noid, and Gregory A Voth. Multiscale modeling of biomolecular systems: in serial and in parallel. *Current opinion in structural biology*, 17(2):192–198, 2007.

[134] Peter J Hunter, Edmund J Crampin, and Poul MF Nielsen. Bioinformatics, multiscale modeling and the iups physiome project. *Briefings in bioinformatics*, 9(4):333–343, 2008.

[135] John C Dallon. Multiscale modeling of cellular systems in biology. *Current Opinion in Colloid & Interface Science*, 15(1):24–31, 2010.

[136] Thomas Eissing, Lars Kuepfer, Corina Becker, Michael Block, Katrin Coboeken, Thomas Gaub, Linus Goerlitz, Juergen Jaeger, Roland Loosen, Bernd Ludewig, et al. A computational systems biology software platform for multiscale modeling and simulation: integrating whole-body physiology, disease biology, and molecular reaction networks. *Frontiers in physiology*, 2:4, 2011.

[137] Jana Schleicher, Theresia Conrad, Mika Gustafsson, Gunnar Cedersund, Reinhard Guthke, and Jörg Linde. Facing the challenges of multiscale modelling of bacterial and fungal pathogen–host interactions. *Briefings in functional genomics*, 16(2):57–69, 2017.

[138] Zhilin Qu, Alan Garfinkel, James N Weiss, and Melissa Nivala. Multi-scale modeling in biology: how to bridge the gaps between scales? *Progress in biophysics and molecular biology*, 107(1):21–31, 2011.

[139] Myong-Hun Chang and Joseph E Harrington Jr. Agent-based models of organizations. *Handbook of computational economics*, 2:1273–1337, 2006.

[140] Carsten Maus, Stefan Rybacki, and Adelinde M Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(1):166, 2011.

[141] Michael W Sneddon, James R Faeder, and Thierry Emonet. Efficient modeling, simulation and coarse-graining of biological complexity with nfsim. *Nature methods*, 8(2):177, 2011.

[142] Nvidia Corporation. Remote visualization on server-class tesla gpus. 2014.

[143] Thomas S Deisboeck. Personalizing medicine: a systems biology perspective. *Molecular systems biology*, 5(1):249, 2009.

[144] Rui Chen and Michael Snyder. Systems biology: personalized medicine for the future? *Current opinion in pharmacology*, 12(5):623–628, 2012.

[145] Yan Li, Mark A Lawley, David S Siscovick, Donglan Zhang, and José A Pagán. Agent-based modeling of chronic diseases: A narrative review and future research directions. *Preventing chronic disease*, 13, 2016.

[146] Canadian institutes of health research personalized medicine. `http://www.cihr-irsc.gc.ca/e/43627.html`. Accessed: 2017-02-16.

[147] Sandro Galea, Matthew Riddle, and George A Kaplan. Causal thinking and complex system approaches in epidemiology. *International journal of epidemiology*, 39(1):97–106, 2009.

[148] Brandon DL Marshall and Sandro Galea. Formalizing the role of agent-based modeling in causal inference and epidemiology. *American journal of epidemiology*, page kwu274, 2014.

[149] Eloise ODonnell, Jo-An Atkinson, Louise Freebairn, and Lucie Rychetnik. Participatory simulation modelling to inform public health policy and practice: Rethinking the evidence hierarchies. *Journal of Public Health Policy*, pages 1–13, 2016.

[150] Gilles Clermont, John Bartels, Rukmini Kumar, Greg Constantine, Yoram Vodovotz, and Carson Chow. In silico design of clinical trials: a method coming of age. *Critical care medicine*, 32(10):2061–2070, 2004.

[151] Rukmini Kumar, Gilles Clermont, Yoram Vodovotz, and Carson C Chow. The dynamics of acute inflammation. *Journal of theoretical biology*, 230(2):145–155, 2004.

[152] Yoram Vodovotz, Carson C Chow, John Bartels, Claudio Lagoa, Jose M Prince, Ryan M Levy, Rukmini Kumar, Judy Day, Jonathan Rubin, Greg Constantine, et al. In silico models of acute inflammation in animals. *Shock*, 26(3):235–244, 2006.

[153] Yoram Vodovotz, Gregory Constantine, James Faeder, Qi Mi, Jonathan Rubin, John Bartels, Joydeep Sarkar, Robert H Squires Jr, David O Okonkwo, Jörg Gerlach, et al. Translational systems approaches to the biology of inflammation and healing. *Immunopharmacology and immunotoxicology*, 32(2):181–195, 2010.

[154] Kama A Wlodzimirow, Saeid Eslami, Robert AFM Chamuleau, Martin Nieuwoudt, and Ameen Abu-Hanna. Prediction of poor outcome in patients with acute liver failuresystematic review of prediction models. *PloS one*, 7(12):e50952, 2012.

[155] James P Boyle, Theodore J Thompson, Edward W Gregg, Lawrence E Barker, and David F Williamson. Projection of the year 2050 burden of diabetes in the us adult population: dynamic modeling of incidence, mortality, and prediabetes prevalence. *Population health metrics*, 8(1):29, 2010.

[156] Theodore Eugene Day, Nathan Ravi, Hong Xian, and Ann Brugh. An agent-based modeling template for a cohort of veterans with diabetic retinopathy. *PloS one*, 8(6):e66812, 2013.

[157] Abdulrahman M El-Sayed, Lars Seemann, Peter Scarborough, and Sandro Galea. Are network-based interventions a useful antiobesity strategy? an application of simulation models for causal inference in epidemiology. *American journal of epidemiology*, page kws455, 2013.

[158] Ross A Hammond and Joseph T Ornstein. A model of social influence on body mass index. *Annals of the New York Academy of Sciences*, 1331(1):34–42, 2014.

[159] Yan Li, Nan Kong, Mark A Lawley, and José A Pagán. Using systems science for population health management in primary care. *Journal of primary care & community health*, 5(4):242–246, 2014.

[160] Gary Hirsch, Jack Homer, Elizabeth Evans, and Ann Zielinski. A system dynamics model for planning cardiovascular disease interventions. *American Journal of Public Health*, 100(4):616–622, 2010.

[161] Yan Li, Nan Kong, Mark Lawley, Linda Weiss, and José A Pagán. Advancing the use of evidence-based decision-making in local health departments with systems science methodologies. *American journal of public health*, 105(S2):S217–S222, 2015.

[162] Mariam Kiran, Paul Richmond, Mike Holcombe, Lee Shawn Chin, David Worth, and Chris Greenough. Flame: simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1633–1636. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[163] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 538–545. IEEE, 2012.

[164] Paul Richmond, Simon Coakley, and Daniela M Romano. A high performance agent based modelling framework on graphics card hardware with cuda. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1125–1126. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[165] Paul Richmond and Mozhgan K Chimeh. Flame gpu: Complex system simulation framework. In *High Performance Computing & Simulation (HPCS), 2017 International Conference on*, pages 11–17. IEEE, 2017.

[166] Roshan M DSouza, Mikola Lysenko, and Keyvan Rahmani. Sugarscape on steroids: simulating over a million agents at interactive rates. In *Proceedings of Agent2007 conference. Chicago, IL*, 2007.

[167] Alcione de Paiva Oliveira and Paul Richmond. Feasibility study of multi-agent simulation at the cellular level with flame gpu. In *FLAIRS Conference*, pages 398–403, 2016.

[168] Shailesh Tamrakar, Paul Richmond, and Roshan M D'Souza. Pi-flame: A parallel immune system simulator using the flame graphic processing unit environment. *Simulation*, 93(1):69–84, 2017.

[169] Nicholson Collier and Michael North. Parallel agent-based simulation with repast for high performance computing. *Simulation*, 89(10):1215–1235, 2013.

[170] Nick Collier. Repast: An extensible framework for agent simulation. *The University of Chicagos Social Science Research*, 36:2003, 2003.

[171] Michael J North, Thomas R Howe, Nick T Collier, and Jerry R Vos. The repast simphony runtime system. In *Proceedings of the agent 2005 conference on generative social processes, models, and mechanisms*, volume 10, pages 13–15. ANL/DIS-06-1, co-sponsored by Argonne National Laboratory and The University of Chicago, 2005.

[172] John T Murphy, Elif Seyma Bayrak, Mustafa Cagdas Ozturk, and Ali Cinar. Simulating 3-d bone tissue growth using repast hpc: Initial simulation design and performance results. In *Winter Simulation Conference (WSC), 2016*, pages 2087–2098. IEEE, 2016.

[173] Thierry Emonet, Charles M Macal, Michael J North, Charles E Wickersham, and Philippe Cluzel. Agentcell: a digital single-cell assay for bacterial chemotaxis. *Bioinformatics*, 21(11):2714–2721, 2005.

[174] Gary R Mirams, Christopher J Arthurs, Miguel O Bernabeu, Rafel Bordas, Jonathan Cooper, Alberto Corrias, Yohan Davit, Sara-Jane Dunn, Alexander G Fletcher, Daniel G Harvey, et al. Chaste: an open source c++ library for computational physiology and biology. *PLoS Computational Biology*, 9(3):e1002970, 2013.

[175] Maciej H Swat, Julio Belmonte, Randy W Heiland, Benjamin L Zaitlen, James A Glazier, and Abbas Shirinifard. Introduction to compucell3d version 3.7. 4. *Detail*, 7(7):7.

[176] Maciej H Swat, Gilberto L Thomas, Julio M Belmonte, Abbas Shirinifard, Dimitrij Hmeljak, and James A Glazier. Multi-scale modeling of tissues using compucell3d. *Methods in cell biology*, 110:325, 2012.

[177] Stefan Hoehme and Dirk Drasdo. A cell-based simulation software for multi-cellular systems. *Bioinformatics*, 26(20):2641–2642, 2010.

[178] Jörn Starruß, Walter de Back, Lutz Brusch, and Andreas Deutsch. Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology. *Bioinformatics*, 30(9):1331–1332, 2014.

[179] Martin Falk, Michael Ott, Thomas Ertl, Michael Klann, and Heinz Koeppl. Parallelized agent-based simulation on cpu and graphics hardware for spatial and stochastic models in biology. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*, pages 73–82. ACM, 2011.

[180] Maciej Cytowski and Zuzanna Szymanska. Large-scale parallel simulations of 3d cell colony dynamics. *Computing in Science & Engineering*, 16(5):86–95, 2014.

[181] Le Zhang, Beini Jiang, Yukun Wu, Costas Strouthos, Phillip Zhe Sun, Jing Su, and Xiaobo Zhou. Developing a multiscale, multi-resolution agent-based brain tumor model by graphics processing units. *Theoretical Biology and Medical Modelling*, 8(1):46, 2011.

[182] Scott Christley, Briana Lee, Xing Dai, and Qing Nie. Integrative multicellular biological modeling: a case study of 3d epidermal development using gpu algorithms. *BMC systems biology*, 4(1):107, 2010.

[183] Amy Henderson, Jim Ahrens, Charles Law, et al. *The ParaView Guide*. Kitware Clifton Park, NY, 2004.

[184] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*, pages 191–198. IEEE, 2005.

[185] Marzia Rivi, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnic. In-situ visualization: State-of-the-art and some use cases. *PRACE White Paper*, pages 1–18, 2012.

[186] Andrew C Bauer, Berk Geveci, and Will Schroeder. The paraview catalyst users guide, 2013.

[187] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick OLeary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on*

In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, pages 25–29. ACM, 2015.

[188] T Kuhlen, R Pajarola, and K Zhou. Parallel in situ coupling of simulation with a fully featured visualization system. 2011.

[189] Yu Su, Yi Wang, and Gagan Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 61–72. ACM, 2015.

[190] Sebastian Grottel, Michael Krone, Christoph Müller, Guido Reina, and Thomas Ertl. Megamol prototyping framework for particle-based visualization. *IEEE transactions on visualization and computer graphics*, 21(2):201–214, 2015.

[191] Mathieu Le Muzic, Ludovic Autin, Julius Parulek, and Ivan Viola. cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets. In *Eurographics Workshop on Visual Computing for Biomedicine*, volume 2015, page 61. NIH Public Access, 2015.

[192] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[193] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. Siam, 2007.

[194] Athan Spiros. Alzheimer's In Silico diffusion of molecules. `http://www.math.ubc.ca/~ais/website/status/diffuse.html`, February 2000.

[195] CUDA NVIDIA. Programming guide, cusparse, cublas, and cufft library user guides.

[196] The VirtualGL Project. VirtualGL background. `http://www.virtualgl.org/About/Background`, 2015.

[197] Geoffrey C Gurtner, Sabine Werner, Yann Barrandon, and Michael T Longaker. Wound repair and regeneration. *Nature*, 453(7193):314, 2008.

[198] API CUDA. Reference manual. 8.0. *NVIDIA. Oct*, 2017.

[199] S Kurita. A comparative study of the layer structure of the vocal fold. *Vocal Fold Physiology*, pages 3–21, 1981.

[200] Mao-Chang Su, Te-Huei Yeh, Ching-Ting Tan, Chia-Der Lin, Oan-Che Linne, and Shiann-Yann Lee. Measurement of adult vocal fold length. *The Journal of Laryngology & Otology*, 116(6):447–449, 2002.

[201] Jaishankar K Kutty and Ken Webb. Tissue engineering therapies for the vocal fold lamina propria. *Tissue Engineering Part B: Reviews*, 15(3):249–262, 2009.

[202] Jean-Michel Prades, Jean Marc Dumollard, Sébastien Duband, Andrei Timoshenko, Céline Richard, Marie Dominique Dubois, Christian Martin, and Michel Peoc'h. Lamina propria of the human vocal fold: histomorphometric study of collagen fibers. *Surgical and Radiologic Anatomy*, 32(4):377–382, 2010.

[203] S Zörner, M Kaltenbacher, and M Döllinger. Investigation of prescribed movement in fluid–structure interaction simulation for the human phonation process. *Computers & fluids*, 86:133–140, 2013.

[204] Pinaki Bhattacharya and Thomas Siegmund. A computational study of systemic hydration in vocal fold collision. *Computer methods in biomechanics and biomedical engineering*, 17(16):1835–1852, 2014.

[205] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, pages 16–21. Boston, MA, 2004.

[206] WDC3D. 6 seamless organic textures 1. `http://wdc3d.com/blog/textures/6-seamless-organic-textures-1/`, April 2010.

[207] Ignacio Castao J.M.P. van Waveren. Real-time normal map dxt compression. `http://www.nvidia.com/object/real-time-normal-map-dxt-compression.html`, February 2008.

[208] L.C. Nttaasen. Beach wood texture. `https://www.flickr.com/photos/magnera/4022717270`, October 2009.

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

[210] Jiska MS Coppoolse, TG Van Kooten, Hossein K Heris, Luc Mongeau, Nicole YK Li, Susan L Thibeault, Jacob Pitaro, Olubunmi Akinpelu, and Sam J Daniel. An in vivo study of composite microgels based on hyaluronic acid and gelatin for the reconstruction of surgically injured rat vocal folds. *Journal of Speech, Language, and Hearing Research*, 57(2):S658–S673, 2014.

[211] Steven F Railsback. Getting results: The pattern-oriented approach to analyzing natural systems with individual-based models. *Natural Resource Modeling*, 14(3):465–475, 2001.

[212] Volker Grimm, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M Mooij, Steven F Railsback, Hans-Hermann Thulke, Jacob Weiner, Thorsten Wiegand, and Donald L DeAngelis. Pattern-oriented modeling of agent-based complex systems: lessons from ecology. *science*, 310(5750):987–991, 2005.

[213] Xinhong Lim, Ichiro Tateya, Tomoko Tateya, Alejandro Muñoz-Del-Río, and Diane M Bless. Immediate inflammatory response and scar formation in wounded vocal folds. *Annals of Otology, Rhinology & Laryngology*, 115(12):921–929, 2006.

[214] Nathan V Welham, Xinhong Lim, Ichiro Tateya, and Diane M Bless. Inflammatory factor profiles one hour following vocal fold injury. *The Annals of otology, rhinology, and laryngology*, 117(2):145–152, 2008.

[215] Tomoko Tateya, Jin Ho Sohn, Ichiro Tateya, and Diane M Bless. Histologic characterization of rat vocal fold scarring. *Annals of Otology, Rhinology & Laryngology*, 114(3):183–191, 2005.

[216] Ichiro Tateya, Tomoko Tateya, Xinhong Lim, Jin Ho Sohn, and Diane M Bless. Cell production in injured vocal folds: a rat study. *Annals of Otology, Rhinology & Laryngology*, 115(2):135–143, 2006.

[217] Tomoko Tateya, Ichiro Tateya, Jin Ho Sohn, and Diane M Bless. Histological study of acute vocal fold injury in a rat model. *Annals of Otology, Rhinology & Laryngology*, 115(4):285–292, 2006.

[218] S Cockbill. The healing process. *Hospital Pharmasist London*, 9(9):255–260, 2002.

[219] Paul Martin. Wound healing–aiming for perfect skin regeneration. *Science*, 276(5309):75–81, 1997.

[220] Martin C Robson, David L Steed, and Michael G Franz. Wound healing: biologic features and approaches to maximize healing trajectories. *Current problems in surgery*, 38(2):A1–140, 2001.

[221] Maria B Witte and Adrian Barbul. General principles of wound healing. *Surgical Clinics of North America*, 77(3):509–528, 1997.

[222] Dianhua Jiang, Jiurong Liang, and Paul W Noble. Hyaluronan in tissue injury and repair. *Annu. Rev. Cell Dev. Biol.*, 23:435–461, 2007.

[223] Robert Stern, Akira A Asari, and Kazuki N Sugahara. Hyaluronan fragments: an information-rich system. *European journal of cell biology*, 85(8):699–715, 2006.

[224] Tracey A Dechert, Ashley E Ducale, Susan I Ward, and Dorne R Yager. Hyaluronan in human acute and chronic dermal wounds. *Wound Repair and Regeneration*, 14(3):252–258, 2006.

[225] George W Bluman. On the transformation of diffusion processes into the wiener process. *SIAM Journal on Applied Mathematics*, 39(2):238–247, 1980.

[226] James B Anderson. A random-walk simulation of the schrödinger equation: H+ 3. *The Journal of Chemical Physics*, 63(4):1499–1503, 1975.

[227] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*, 22:27–46, 2016.

[228] Amanda Randles, Erik W Draeger, and Peter E Bailey. Massively parallel simulations of hemodynamics in the primary large arteries of the human vasculature. *Journal of computational science*, 9:70–75, 2015.

[229] AV Husselmann and KA Hawick. Spatial agent-based modelling and simulations-a review. *CSTN Computational Science Technical Note*, 153, 2011.