EDITORS: Jeffrey C. Carver, carver@cs.ua.edu
Karla Morris, knmorri@sandia.gov

SPECIAL TRACK: SOFTWARE ENGINEERING

OpenACC Acceleration of an Agent-Based Biological Simulation Framework

Matt Stack, *NVIDIA Corporation, Santa Clara, CA, 95051, USA*
Paul Macklin, *Indiana University, Bloomington, IN, 47408, USA*
Robert Searles, *NVIDIA Corporation, Santa Clara, CA, 95051, USA*
Sunita Chandrasekaran, *University of Delaware, Newark, DE, 19716, USA*

Computational biology has increasingly turned to agent-based modeling to explore complex biological systems. Biological diffusion (diffusion, decay, secretion, and uptake) is a key driver of biological tissues. GPU computing can vastly accelerate the diffusion and decay operators in the partial differential equations used to represent biological transport in an agent-based biological modeling system. In this article, we utilize OpenACC to accelerate the diffusion portion of PhysiCell, a cross-platform agent-based biosimulation framework. We demonstrate an almost 40× speedup on the state-of-the-art NVIDIA Ampere 100 GPU compared to a serial run on AMD's EPYC 7742. We also demonstrate 9× speedup on the 64-core AMD EPYC 7742 multicore platform. By using OpenACC for both the CPUs and the GPUs, we maintain a single source code base, thus creating a portable yet performant solution. With the simulator's most significant computational bottleneck significantly reduced, we can continue cancer simulations over much longer times.

Computational biology has increasingly turned to agent-based modeling—which represents individual biological cells as discrete software agents—to explore complex biological systems where many cells interact through the exchange of mechanical forces, exchange of diffusing chemical factors, and other biomechanical feedback. Biological diffusion (diffusion, decay, secretion, and uptake) is a key driver of biological tissues. Blood vessels release nutrients (oxygen, glucose, and other key metabolites) that diffuse through tissues to be consumed by cells and then absorb diffusible waste products, while cells secrete and absorb diffusible chemical factors to communicate and coordinate their behaviors.[1] See Figure 1 for a typical 3-D simulation model.

1521-9615 © 2022 IEEE
Digital Object Identifier 10.1109/MCSE.2022.3226602
Date of current version 1 May 2023.

Therefore, most modern agent-based biological modeling systems are hybrid: they combine discrete cell agents with partial differential equations (PDEs) to represent biological transport, such as

$$\frac{\partial \boldsymbol{\rho}}{\partial t} = \boldsymbol{D}\nabla^2\boldsymbol{\rho} - \boldsymbol{\lambda}\boldsymbol{\rho} + \sum_{\text{cells } i} \delta(\boldsymbol{x} - \boldsymbol{x}_i)V_i(\boldsymbol{S}_i(\boldsymbol{\rho}_i^* - \boldsymbol{\rho}) - \boldsymbol{U}_i\boldsymbol{\rho}) \tag{1}$$

where $\boldsymbol{\rho}$ is a vector of diffusible substrates, and each cell agent *i* has position $\boldsymbol{x}_i$ and volume V_i, a vector of secretion rates $\boldsymbol{S}_i$ and uptake rates $\boldsymbol{U}_i$, and a "target" extracellular substrate vector $\boldsymbol{\rho}^*i$. (Vector–vector products are taken elementwise, and δ is the Dirac delta function.) Numerical stability and accuracy require that these PDEs be solved with relatively small step sizes Δt, making the solution of biological diffusion PDEs a rate-limiting step in hybrid agent-based biological models that can limit the maximum size and duration of simulations. This, in turn, can hinder high-throughput simulation model exploration (e.g., newer model calibration techniques like

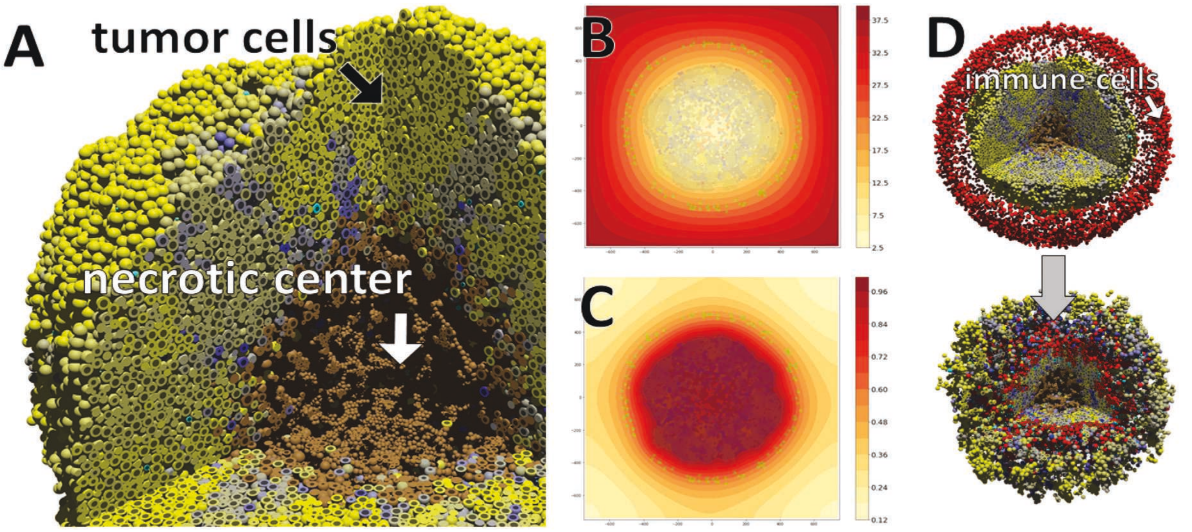

FIGURE 1. Typical PhysiCell model. The "`cancer_immune_sample`" is a typical example of a complex biological system modeled with the PhysiCell framework. (a) Tumor cells (colored from blue to yellow by aggressiveness) grow, divide, and die based upon the local availability of oxygen that diffuses from the computational boundary. (b) The tumor cell consumption of oxygen leads to the formation of oxygen gradients and eventual necrosis (death) in the center (brown cells). (c) Tumor cells also release a diffusible immunostimulatory factor that attracts immune cells (red), which (d) move by biased random migration toward tumor cells, attach, and preferentially kill highly immunogenic tumor cells (yellow). Readers can interactively run a 2-D version of this model in a web browser at https://nanohub.org/tools/pc4cancerimmune.

approximate Bayesian computation), which require running thousands or millions of simulations quickly.

GPU computing can vastly accelerate the diffusion and decay operators in (1), but the tight coupling with cell-based sources—which dynamically move and change their sizes and rate constants—makes the overall solution more challenging. Moreover, the number of discrete cell agents can change by orders of magnitude over the course of a cancer simulation. Recent examples[2] have confirmed the potential for GPU-accelerated agent-based models but, to date, have generally been limited to simplified systems that do not include complex intracellular-level dynamics or complex cell–cell interactions and other rules.

In this article, we utilize OpenACC, a directive-based programming model, to accelerate the diffusion portion in (1) in PhysiCell,[3] a cross-platform agent-based biosimulation framework that has been adopted in cancer,[4] infectious diseases,[5] and other complex biological problems. Prior to this work, PhysiCell had been optimized for multicore simulations with OpenMP. With default settings, PhysiCell advances the numerical solution of (1) with 10 diffusion steps (Δt_{diff}) before advancing the cell–cell mechanical interactions by one mechanics step (Δt_{mech}); cell biological processes (e.g., cell cycle progression, death, and "decision making") are advanced on a much slower cell timescale of $\Delta t_{\text{cell}} \sim 60\Delta t_{\text{mech}}$. This separation of timescales allowed us to prioritize GPU optimization to the biological diffusion solver.

Using the NVIDIA high-performance computing (HPC) software developer's kit (SDK) OpenACC 21.3, we demonstrate an almost 40$\times$ speedup using managed memory (where data movement between the host and the device is handled automatically by the underlying CUDA runtime) on the state-of-the-art NVIDIA Ampere 100 (A100) GPU compared to a serial AMD EPYC core 7742 for a 360-simulated-minute input dataset. We also demonstrate 9$\times$ speedup on the 64-core AMD EPYC 7742 multicore platform using NVIDIA HPC SDK OpenACC 21.3. By using OpenACC for both the CPUs and the GPUs, we maintain a single source code base, thus creating a portable yet performant solution.

This is a critical step toward a portable GPU acceleration of a detailed agent-based simulation platform for complex biological systems. With the simulator's most significant computational bottleneck significantly reduced, we can look toward continuing cancer simulations over much longer times. It also represents a key step toward converting this code from purely CPU based toward an MPI+X paradigm. In doing so, large 3-D simulation domains are decomposed into smaller subdomains residing on individual HPC nodes with message passing interface (MPI) data exchange. In each compute node's subdomain, cell "logic" continues to be parallelized for multicore SMP with OpenMP, and critical low-level mechanics like biological diffusion and cell–cell mechanical interactions are parallelized on the GPU (or other specialized accelerators) via OpenACC. This will help grow the code to allow 3-D multiscale simulations of cancer and diseases in unprecedented detail on emerging leadership-class HPC resources: a critical step toward advancing cancer patient digital twins that can simulate clinically significant tumors fast enough to be clinically actionable for patient care decisions.

DESIGN AND IMPLEMENTATION

This section describes the design and implementation choices we employed while parallelizing and accelerating PhysiCell on a heterogeneous system.

Directive-Based Programming With OpenACC

OpenACC is a performance-portable, directive-based parallel programming model that targets modern heterogeneous HPC hardware. Several real-world applications and top HPC applications use OpenACC and for the same reasons we felt confident in using OpenACC for PhysiCell. These OpenACC applications include ANSYS,[a] Gaussian,[b] COSMO,[c] and many more. Please refer to this tracker[d] where OpenACC has been collecting published OpenACC papers. Given the number of successful OpenACC ports of applications, we chose to go with OpenACC, given its easier adaptability, stability, and maturity, instead of the OpenMP offloading model. Based on our experience, the latter still has room for improvement before it could be more easily adopted for complex applications like PhysiCell. Having said that, as part of our continued efforts on PhysiCell in the near future, we plan to explore the OpenMP offloading model as well.

Using the OpenACC Programming Model for PhysiCell

PhysiCell-GPU is designed to run the diffusion portion of PhysiCell in an accelerated mode. More details are available in an article by Ghaffarizadeh *et al.*[6] BioFVM simulates the diffusive transport of substrates, while PhysiCell simulates the cells.[3] BioFVM has been parallelized on multicore systems using a directive-based programming model, OpenMP. While parallelization on several cores can be beneficial, we hypothesized that the move to the GPU would see even greater performance in key sections of the code that would benefit from being accelerated on GPUs.

To test this, we explored the usage of OpenACC to achieve performance while maintaining cross-platform portability in BioFVM. Irrespective of using OpenACC or OpenMP, the fundamental idea of a directive-based programming model for GPUs includes taking an approach similar to the low-level programming framework for GPUs, such as CUDA, where data needed for compute is passed to the GPU; the parallel functions are computed on the GPU; and then, finally, the updated data are pulled back to the host for further processing.

ADO ALLOWS FOR CONTINUOUS PERFORMANCE PROGRESS BY RE-EVALUATING HOT SPOTS AT EACH STAGE OF THE PORTING CYCLE WITH PROFILING.

Profiling: Identifying Hot Spots

We initially profiled the "`cancer_immune_sample`" sample project that is bundled with every PhysiCell download.[3] This 3-D sample is representative of (CPU-based) PhysiCell models: the model includes multiple diffusible substrates (oxygen and an inflammatory factor) as well as multiple cell types (red immune cells attack tumor cells), heterogeneous cell properties (blue tumor cells are less aggressive and less immunogenic; yellow tumor cells are more aggressive but also more immunogenic), and customized cell–cell mechanical interaction rules (immune cells seek tumor cells, test for contact, adhere with spring-like terms, test tumor cell properties, and probabilistically induce death). (See Figure 1.) This profiling was used to identify the hot spots that would be candidates for acceleration. The NVPROF command-line profiler confirmed the insight that diffusion was the dominating portion of the total runtime of the original CPU-only code. Figure 2 shows that the diffusion function dominated the total time and, hence, was our target of optimization. For profiling, we used NVPROF and the NVIDIA Nsight Systems profiler. Additionally, we used a technique called analysis-driven optimization (ADO) to profile the code at the start and then after porting a given section to the GPU. ADO allows for continuous performance progress by re-evaluating hot spots at each stage of the porting cycle with profiling.

[a]shorturl.at/lC128
[b]https://on-demand.gputechconf.com/gtc/2016/presentation/s6524-roberto-gomperts-enabling-the-electronic.pdf
[c]https://www.openacc.org/success-stories/cosmo
[d]shorturl.at/mquv0

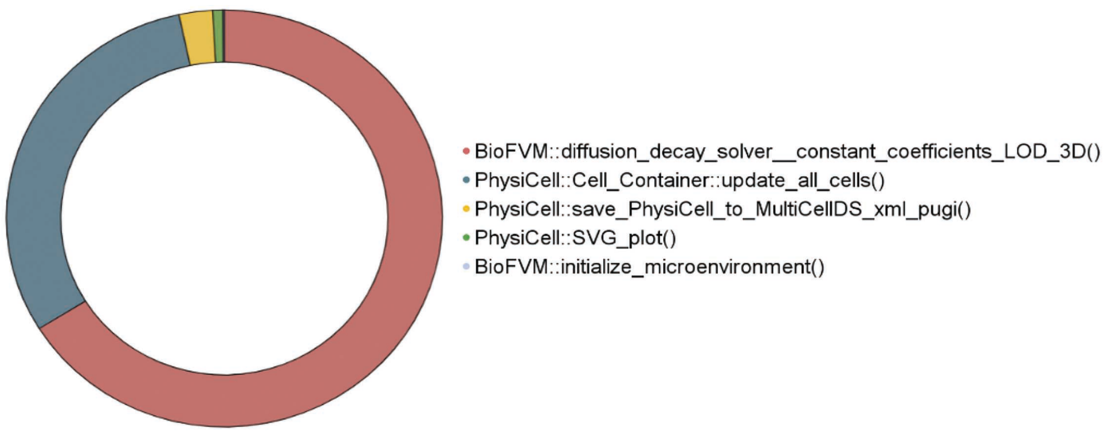

FIGURE 2. Initial code profiling. An analysis with NVProf identified that approximately two thirds of execution time was spent on biological diffusion (blue), with evaluation of the discrete cell agent mechanical and biological rules (red) also requiring significant computation time.

Elements of Parallelism

In PhysiCell, diffusion involves a number of computational steps and function calls. The data invoked by these functions is handled with care to ensure a good computation-versus-memory transfer balance. After initialization of the GPU copy of *p-density-vectors* in GPU memory, updates are only processed to the host at the user's request (user manually managed memory) or when there is a page fault on the data initiated by the host (OpenACC managed memory). Because the data are ported to arrays, each part of the diffusion section is carefully mapped from its original state of operating on vectors to operating on arrays. The full code corresponding to this article is available on GitHub at https://github.com/matt-stack/PhysiCell_GPU, branch PhysiCell_GPU_Stable.

The flow begins in file BioFVM/BioFVM_solvers.cpp, which acts as the jumping point into memory transfers and launching calculations. The functions called in BioFVM/BioFVM_solvers.cpp are implemented in BioFVM/BioFVM_microenvironment.cpp and handle the OpenACC code. Please see Figure 3 for a visual representation.

Data *p-density-vectors* is the main array for the data that would be updated from the diffusion functions and transferred between device and host. Originally, this data structure is a "vector of vectors" or a 2-D vector (where each voxel in the simulation grid holds a vector of diffusible substrate values). The *p-density-vector* on the host is std::vectors, and the *p-density-vector-GPU* on the device are arrays. The functions *transfer-3-D()* and *transfer-2-D()* are called to initialize the memory space on the GPU and to copy the data currently in *p-density-vectors*. The functions *translate-vector-to-array()* and *translate-array-to-vector()* translate the data back and forth between the array and vector

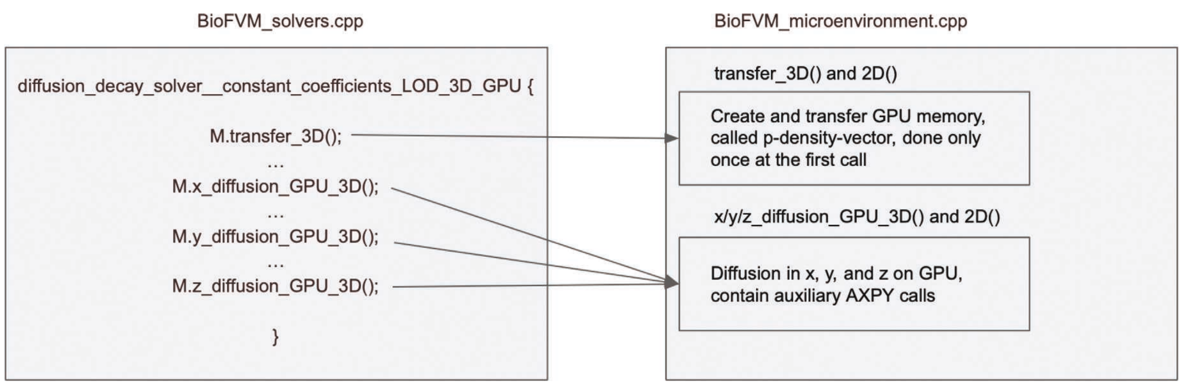
BioFVM_solvers.cpp
diffusion_decay_solver__constant_coefficients_LOD_3D_GPU {
M.transfer_3D();
...
M.x_diffusion_GPU_3D();
...
M.y_diffusion_GPU_3D();
...
M.z_diffusion_GPU_3D();
}
BioFVM_microenvironment.cpp
transfer_3D() and 2D()
Create and transfer GPU memory, called p-density-vector, done only once at the first call
x/y/z_diffusion_GPU_3D() and 2D()
Diffusion in x, y, and z on GPU, contain auxiliary AXPY calls

FIGURE 3. Diagram of the code flow and files.

LISTING 1. The OpenACC data flow between the host and device.

```
#pragma acc enter data create
#(this->temp_physicell_array[0:bin_physicell_array][0:0])
    for (int i = 0; i < bin_physicell_array; i ++) {
            int sze = physicell_array[i].size();
            temp_physicell_array[i] =
            physicell_array[i].data();
            #pragma acc enter data copyin
            #(this->temp_physicell_array[i:1][:sze])
    }
```

types on the host. This is used after the host-side mirror of the GPU array data is updated to the host. If there is any work done in *p-density-vectors* on the host, then these methods will convert the data to array form and initiate the update to the device.

The rest of the data needed by the diffusion section are initialized in the *transfer-3-D()* and *transfer-2-D()*. These data are essential to the computation and are converted to array form like *p-density-vectors*, but the data are not typically updated back and forth between host and device. If the need arises, a function could easily be made to create a host-side mirror and OpenACC directives to handle the memory updates.

The pseudocode in Listing 1 shows an OpenACC code that handles data flow between the device and host. This example shows an array of arrays, physicell_array, which undergoes a deep copy; each inner array gets allocated on the device while keeping the pointers intact so that the outer array can be referenced. The *this* keyword is crucial to the process, as that lets the compiler know to keep the newly created data on the device interlocked with the surrounding data.

Compute *x-diffusion-GPU-2-D()* exists for the direct computation step for diffusion. In 2-D, this is performed in the X and Y and, in 3-D, in the X, Y, and Z vectors. The algorithm from the original PhysiCell stayed the same, while the syntax was changed to account for the new array format of the crucial data. Slight modifications are made to the original design, including creating a separate axpy-acc function, wrapped in an OpenACC routine directive to enable GPU execution. Storing the specific size of the *p-density-vectors* in GPU memory is important for many loop bounds in the *x-diffusion-GPU-2-D()* section. The *apply-dirichlet-conditions-GPU()* function is an important auxiliary function that was initially not set to need to run on the GPU. We learned that the Dirichlet condition on the GPU eliminated the need for high-rate data updates between the Dirichlet and diffusion computation steps. The original version utilized OpenMP parallel for the top-level loop, and our version kept to a similar organization to adhere to correctness.

The *axpy-acc()* function helps manage the more complex diffusion section by replacing the inline axpy and naxpy (basic linear algebra subroutines) in each *x, y,* and *z* diffusion function with a single function. Abstracting AXPY away from the multiple inline improves readability and promotes a modular design.

Methods of Verification for Data Integrity

We used three methods of verification to ensure the OpenACC implementation maintains the mathematical integrity of the formulas.

Method 1: Convergence Test

Our first validation test method uses a 1-D convergence test from the original BioFVM method article[6] to ensure that the expected numerical accuracy is maintained. Briefly, this tests the ℓ_∞ norm of the numerical solution against a known analytical solution at multiple times for a 1-D problem with diffusion, zero flux (Neumann) boundary conditions, and a nontrivial initial condition. See Ghaffarizadeh *et al.*[6] for further details.

For the more complex 3-D simulation where there is no analytical solution, we computed the solution at multiple times ($t = 60$, 180, and 360 s) using both the accelerated code and the original CPU-based code. The results were bitwise identical, showing that the accelerated code can be fully validated against the original code that has been extensively convergence tested and peer reviewed.[3,7]

Method 2: p-density-vectors *cross check*

In method 2, we compared the data from PhysiCell-GPU against the original CPU-based PhysiCell in each voxel. As noted previously, PhysiCell-GPU uses an array version of the data structure *p-density-vector* to represent the microenvironment data on the GPU. PhysiCell originally used C++ std::vectors, which have an issue porting easily to the GPU. In the past, there has been a limited implementation of std::vector by the PG Group, but this implementation would not work in this case due to the complexity of PhysiCell's *p-density-vector* data structure and the microenvironment C++ class structure. The *p-density-vector* is a pointer to a `std::vector<std::vector<double>>`, and this extra layer prevents us from utilizing the feature. PhysiCell-GPU has a number of differences we needed to ensure we did not introduce any difference against the reference. Testing for compiler, compile flag, and architecture differences was important to assess PhysiCell-GPU.

Method 3: Visual inspection of output

Methods 1 and 2 gave quantitative ways to validate PhysiCell-GPU against simple examples with known analytic solutions. We devised method 3 to give a test against more realistic conditions in a higher dimensional geometry that more closely replicates typical modeling conditions but where analytic solutions are not available. To verify the success of the port in terms of keeping full integrity of the data after going through diffusion on the GPU, visual inspection was used on the output for a fast yet reliable method. We had created a Python script to display the necessary data from the microenvironment on a plot. By design, a small change in functional performance would cause a dramatic change in cell data and, therefore, the visualization, thus helping us to readily identify errors.

Figure 4 shows an example of vastly different outputs caused by a simple "off-by-one" error in the Dirichlet function; Figure 5 shows the same visual inspection test after the bug was fixed. While techniques (noted earlier) were used that inspected every element, the visual inspection method was often used before any other as a glaring indicator of data corruption.

EXPERIMENTAL SETUP AND INPUT DATASETS

This section highlights the experimental setup details for the results. We have primarily used two NVIDIA DGX machines. The hardware and software details are described in Table 1.

While the original PhysiCell code was compiled using OpenMP-enabled *gcc* version 7.5.0, we used the NVIDIA HPC SDK's *nvc* 21.3 compiler for parallelizing and accelerating the code. This compiler supports both OpenMP parallelization and OpenACC offloading.

Input Datasets

PhysiCell input is written in XML. In the original PhysiCell code, the parameters changed were x min, x max, y min, y max, z min, z max, max time, and omp num threads. The set of results came from x, y, and z min set at -1000 along with x, y, and z max set at 1000 with the max time units at 60 simulated minutes. The parameter omp num threads was set at 1 and 32, representing a serial implementation and an optimal performance CPU thread count, respectively. The second set of results kept all parameters the same except increasing the max time units to 180 simulated time, and a third

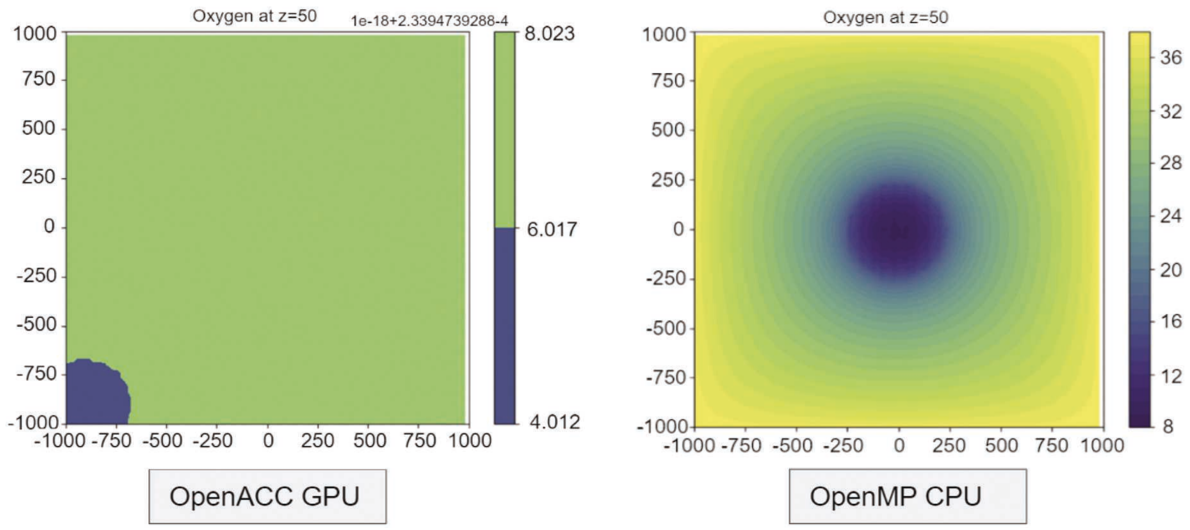

FIGURE 4. Method 3 validation shows a drastically different visual appearance in the presence of an "off-by-one" bug.

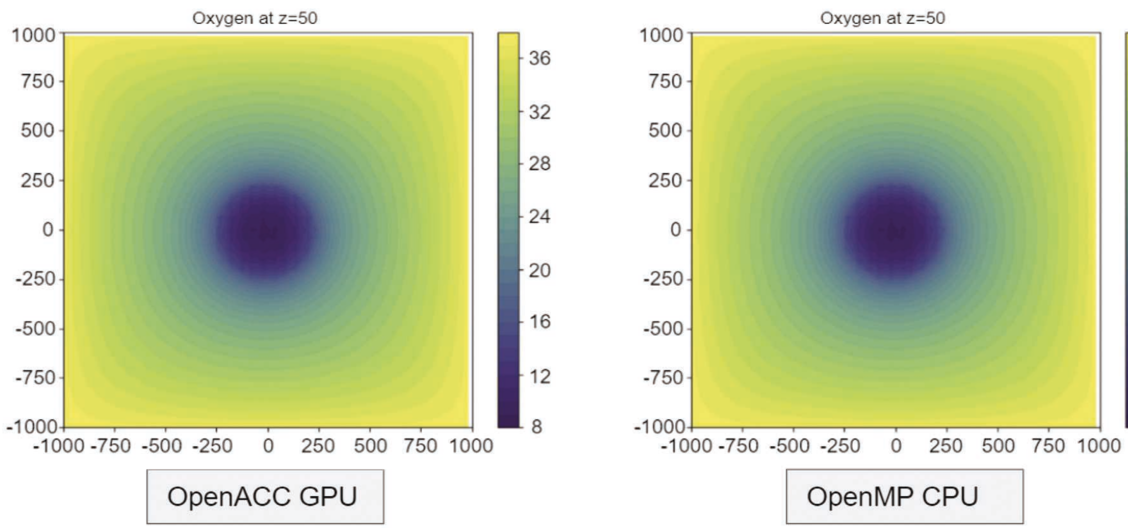

FIGURE 5. Method 3 validation shows identical images when the implementation bugs have been fixed.

for 360 simulated minutes. This input set represents a cell with a size of 1 cm.

RESULTS

This section presents the results on the CPUs and the GPUs. Table 2 presents results for a single-core CPU, 64-core CPU, NVIDIA's V100 GPU, and NVIDIA's A100 GPU using two implementations: namely, the manual and the managed-memory implementations for the different input datasets.

The OpenMP CPU results use NVIDIA's NVHPC 21.3 showing the time taken using 1 thread (open CPU with one core) on AMD EPYC Rome to simulate a serial version. Using one thread is close to running the code in the native serial manner, but it is not exact, as the code has been optimized from the native serial version. We were forced to take this approach because a native serial version of the code is not available. We then show the runtime using 64 cores (OMP CPU with 64 cores) to demonstrate the speedup of approximately 7.9× when running the OpenMP version on a high-end CPU compared to the one-core serial version. Figure 6 shows that a speedup of almost 40× using OpenACC to target an NVIDIA A100 GPU using managed memory was achieved over the serial CPU configuration for 360 simulated minutes as the dataset. Overall, the newer and more powerful NVIDIA A100 GPU shows a better speedup with the same code over the V100 GPU, which is consistent with our expectations.

Table 2 shows the execution time of the manual and the managed-memory implementations for the GPUs. In our case, the managed-memory implementation has the benefit of less memory transfer time compared to manual memory implementation in exchange for less fine-grain control of the GPU resident data. Figure 6 shows the results in the form of relative speedups. *(Manual GPU A100)* and *(Managed GPU A100)* have the same compute time, but *(Managed GPU A100)* spends less time transferring memory. They show that, while data transfer from the CPU to the GPU is relatively expensive up front, the benefit is seen over more time steps with more realistic simulation lengths. This is due to the fact that data can remain on the GPU in between time steps, which reduces the amortized cost of the up-front data transfers and increases the benefit of the accelerated diffusion as the number of time steps is increased.

Figure 7 shows an Nsight Systems profile of the code running on the GPU using managed memory. We observe the large up-front data transfer cost in green, followed by the computation that makes up each time step in blue. Note that there are no large pieces of data movement to or from the GPU in between the time

TABLE 1. Specifications of the nodes in the two systems.

Machine	CPU	NVIDIA GPU
NVIDIA DGX-2	Intel Xeon Platinum 8168 (24 cores)	Volta V100 (32 GB HBM2)
NVIDIA DGX A100	AMD EPYC Rome 7742 (64 cores)	Ampere A100 (40 GB HBM2)

TABLE 2. Results.

Simulation Dataset	60 Simulation Minutes (s)	180 Simulation Minutes (s)	360 Simulation Minutes (s)
OMP CPU 1 Core	524.6083	1,511.1268	3,107.043
OMP CPU 64 Cores	66.0669	201.9457	404.9028
ACC CPU 64 Cores	57.993	167.4116	330.3994
Manual GPU V100	94.2378	159.4965	257.9657
Manual GPU A100	92.0517	151.6884	241.1578
Managed GPU V100	23.903	57.4191	107.7914
Managed GPU A100	21.3251	45.9034	82.7607

steps shown. Overall, we see speedups that benefit the domain application by allowing larger problem sizes and longer simulation times in less wall time.

This is the first work to apply portable GPU acceleration (via OpenACC) to biological diffusion in PhysiCell—a critical bottleneck to larger and longer simulations. Others have recently applied MPI accelerations to BioFVM[7] and PhysiCell to advance toward billion-cell simulations.[8] However, these require significant code refactoring and HPC resources to attain their full performance. They have not been optimized to leverage workstation (or single-compute-node) GPU resources. Ultimately, a hybrid OpenACC–MPI architecture could attain still better performance by leveraging our OpenACC-based GPU acceleration on individual compute nodes, with MPI and domain decomposition to attain billion-cell scalability.

GPU-Accelerated Long-Time Simulations

To assess the potential scientific impact of the GPU-accelerated code, we performed tests on PhysiCell's built-in `cancer_immune_sample` sample project that originated in the work by Ghaffarizadeh *et al.*[3] (See the brief description of this sample project in Figure 1.) To maintain our focus on the GPU acceleration of the diffusion code, we disabled cell mechanics and phenotype changes, thus leaving cell positions, sizes, and behaviors fixed. To emulate a typical scientific workflow, we performed a full data save every seven simulated days. In the original work[3] and subsequent 3-D parameter space investigations,[9] simulations were limited to 21 days (three data checkpoints) due to CPU limitations, each requiring approximately 48 h (wall time) to complete. Based on our profiling, we estimate that diffusion consumed approximately 65% (~30 h) of the wall time for each simulation in prior studies.

After recompiling the CPU-based code with the same `nvc++` compiler, we simulated the 3-D model (diffusion only) with default parameter values on an AMD EPYC 64-core CPU. The CPU-only code only reached the first checkpoint (seven days) and was not able to complete the full 21-day simulation within the maximum 4-h time limit on the DGX system (a typical constraint on large-scale model exploration on shared systems); based on the simulation's progress by this time limit and prior benchmarking,[6] we estimate that the CPU-only code would require approximately 9.3 h (wall time) to complete the full 21-day simulation on the system. Only the first checkpoint (equal to a seven-day simulation) finished at 3 h, 6 min, and 4 s of wall

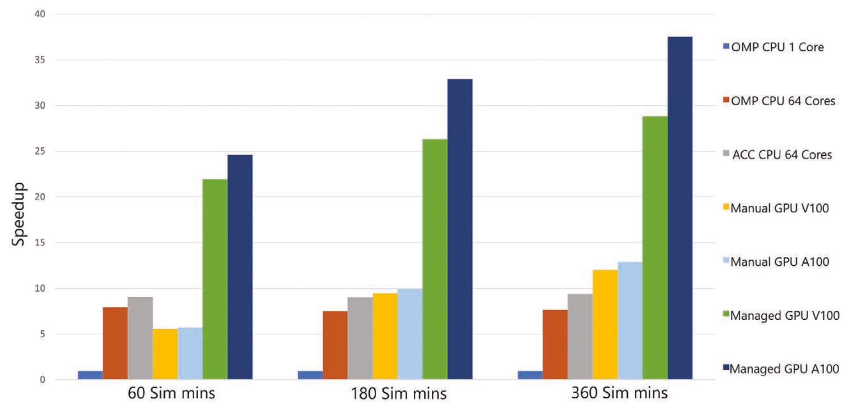

FIGURE 6. Speedup normalized over serial.

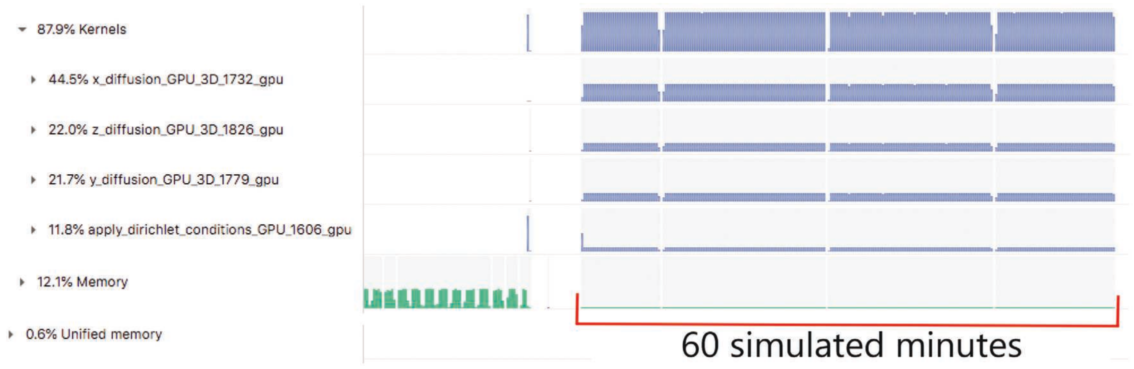
87.9% Kernels
44.5% x_diffusion_GPU_3D_1732_gpu
22.0% z_diffusion_GPU_3D_1826_gpu
21.7% y_diffusion_GPU_3D_1779_gpu
11.8% apply_dirichlet_conditions_GPU_1606_gpu
12.1% Memory
0.6% Unified memory
60 simulated minutes

FIGURE 7. NVIDIA Nsight systems profile of Physicell_GPU with managed memory. The green bars indicate the up-front data transfer cost. The blue blocks indicate the computation/time step. Although composed of individual kernel launches, the red bracket shows a zoomed-out view of all of the GPU work for this test case.

time. In comparison, the OpenACC GPU-accelerated code (managed memory and A100) completed the entire 21-day simulation (three checkpoints) in 1 h, 42 min, and 13.52 s.

We next asked how far an OpenACC GPU-accelerated simulation (managed memory and A100 GPU) could be extended within a maximum 4-h time limit by increasing the test's maximum simulation time and continuing to save data every seven simulated days. Whereas the CPU-only code (64 cores and AMD EPYC) only reached the first seven-day checkpoint, the OpenACC code was able to reach and pass the 42-day checkpoint (six checkpoints), requiring 3 h and 24 min to simulate 42 days of wall time. By contrast, we estimate that CPU-only code would have required more than 18 h to complete the same computations. This shows that using the GPU-accelerated diffusion code should enable long-duration simulations (e.g., of cancer and disease progression) that were previously only feasible on systems that permit submitted jobs to run for hours or days. However, many shared computational resources are not available for the length of time that would be required. Without accelerated computing, a timeout of resources would be reached long before the desired results were computed.

This is particularly relevant for the emerging field of cancer forecasting and cancer patient digital twins: after calibration, clinical teams will use the patient's digital twin to simulate thousands of candidate treatment plans (virtual control, standard of care, and multiple combinations of therapies under a variety of dosing options) over weeks or months, each with tens of replicates to estimate uncertainty. These multiweek simulations must execute quickly to allow timely clinical decision support; supporting clinical workflows at scale will further increase the need for rapidly executing long-duration forecasting simulations.

CONCLUSION

Code profiling of a complex, multiscale biological agent-based code revealed that, for typical simulation models, approximately 65% of the execution time (wall time) is spent on biological diffusion, making this a logical first target for GPU optimization. Offloading numerical operations to solve the diffusion along with careful management of memory transfers between host and device memory resulted in an approximately 40-fold reduction in the execution time for the biological diffusion solver. Multiple testing methods were used to validate the GPU code against the original CPU code.

In real-world testing against a complex 3-D cancer immunology example, switching from a 64-core CPU implementation to a managed-memory OpenACC GPU implementation reduced the execution time (wall time) for 21 days of biological diffusion and data saves from 9.3 h to 1.7 h—a reduction of more than 80%. Based on earlier profiling that diffusion is responsible for more than 65% of total execution time, we estimate that full simulation code—once again, including the cell mechanics and biological calculations responsible for the remaining 35% of execution time—would require approximately 50% less execution time once using the GPU-accelerated diffusion algorithms developed in this article. Furthermore, the GPU-accelerated code was able to complete a diffusion-only simulation twice as long as the CPU-based code in roughly one third of the time, allowing us to simulate further in time than was previously feasible while observing new biological dynamics.

A full 21-day simulation might be expected to take $\sim\frac{1}{0.65}\times 9.3$ h (14–15 h) to execute, so the GPU-accelerated full model should see its total wall time reduced to 7–8 h. This could potentially allow us to double the duration of each simulation for the same amount of execution time, thus exposing new biological dynamics. This confirms the scientific benefit of accelerating targeted portions of the code with careful GPU optimization.

FUTURE WORK

The current implementation yielded a significant performance increase that, in turn, “unlocked” new scientific possibilities in the code, particularly longer-time simulations. However, more could be done. First, the OpenACC acceleration of the biological diffusion still relies upon CPU operations for the cell-based secretion and uptake of biological substrates. These may account for the difference between the 40$\times$ speedup in simpler performance benchmarking and the “real-world” 3-D test (5–6$\times$ speedup) where cell secretion and uptake play a greater role. Moving these operations on the GPU could further reduce the need for costly memory transfers, allowing all computations to “reside” on the device for 10 computational diffusion steps (with size $\Delta t_{\text{diff}}\sim 0.01\ \text{min}$) without the need for memory transfer. Second, cell mechanics operations are governed by biased random migration and interaction potentials, which are well suited to GPU computations.[10] Furthermore, there are 60 mechanics steps (with step size $\Delta t\text{mech}\sim 0.1\ \text{min}$) for every cell step (with size $\Delta t_{\text{cell}}\sim 6\ \text{min}$). Thus, moving cell mechanics solvers would not only accelerate those computations but drastically reduce the need for host–device memory transfers: computations could “reside” on the device for 600 computational steps before transferring data to the host memory.

Future work will explore these refinements. Biological diffusion—which presents work accelerated by a factor of 40—accounts for approximately 65$\%$ of the execution time in typical simulations. Cell updates—dominated by 10 mechanics steps for every biological step—accounts for another 25–30$\%$ of the execution time. If the mechanics code (approximately 25$\%$ of original execution time) could be similarly accelerated by a factor of 40, then the overall simulation execution time should be reduced on the order of 85–90$\%$: a substantial speedup.

A 10–100$\times$ speedup would enable exciting new scientific possibilities. If simulation sizes were left unchanged, then the speedup would enable much faster execution times for individual simulations; this is critical for data assimilation and parameter estimation techniques, like approximate Bayesian computing, that must rapidly run many simulations sequentially. Similarly, leaving the simulation size unchanged, we could simulate to longer times, which will be of great use to digital twin efforts in medicine. If execution time were left unchanged, then larger domains with more agents could be simulated, enabling studies of more complex tissues and even small organisms. Finally, the increased “computational budget” afforded by a more complete GPU acceleration could allow us to introduce multiple agents per biological cell, allowing sophisticated simulations of not only cell morphology (e.g., as in subcellular element models[11]), but even introducing agents for subcellular components and biophysical processes, such as the movement, fission, and fusion of Golgi bodies during signaling.[12] This could be transformative in relating emerging high-resolution microscopy to intracellular biophysics and functional biology.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant 1814609.

REFERENCES

1. P. Macklin *et al.*, “Progress towards computational 3-D multicellular systems biology,” in *Systems Biology of Tumor Microenvironment*, K. Rejniak, Ed. Cham, Switzerland: Springer International Publishing, 2016, ch. 12, pp. 225–246.
2. P. Germann, M. Marin-Riera, and J. Sharpe, “ya||a: GPU-powered spheroid models for mesenchyme and epithelium,” *Cell Syst.*, vol. 8, no. 3, pp. 261–266, Mar. 2019, doi: 10.1016/j.cels.2019.02.007.
3. A. Ghaffarizadeh, R. Heiland, S. H. Friedman, S. M. Mumenthaler, and P. Macklin, “PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems,” *PLoS Comput. Biol.*, vol. 14, no. 2, Feb. 2018, Art. no. e1005991, doi: 10.1371/journal.pcbi.1005991.
4. H. L. Rocha *et al.*, “A persistent invasive phenotype in post-hypoxic tumor cells is revealed by novel fate-mapping and computational modeling,” *iScience*, vol. 24, no. 9, Aug. 2021, Art. no. 102935, doi: 10.1016/j.isci.2021.102935.
5. M. Getz *et al.*, “Iterative community-driven development of a SARS-CoV-2 tissue simulator,” Nov. 2021, doi: 10.1101/2020.04.02.019075v5.
6. A. Ghaffarizadeh, S. H. Friedman, and P. Macklin, “BioFVM: An efficient, parallelized diffusive transport

solver for 3-D biological simulations," *Bioinformatics*, vol. 32, no. 8, pp. 1256–1258, Apr. 2016, doi: 10.1093/bioinformatics/btv730.

7. G. Saxena, M. Ponce-de Leon, A. Montagud, D. V. Dorca, and A. Valencia, "BioFVM-X: An MPI+OpenMP 3-D simulator for biological systems," in *Computational Methods in Systems Biology*, E. Cinquemani and L. Paulevé, Eds. Cham, Switzerland: Springer International Publishing, 2021, pp. 266–279.
8. A. Montagud, M. Ponce de León, and A. Valencia, "Systems biology at the giga-scale: Large multiscale models of complex, heterogeneous multicellular systems," *Current Opinion Syst. Biol*., vol. 28, Dec. 2021, Art. no. 100385, doi: 10.1016/j.coisb.2021.100385.
9. J. Ozik *et al*., "High-throughput cancer hypothesis testing with an integrated PhysiCell-EMEWS workflow," *BMC Bioinformat*., vol. 19, no. S18, Dec. 2018, Art. no. 483, doi: 10.1186/s12859-018-2510-x.
10. E. Wright, M. H. Ferrato, A. J. Bryer, R. Searles, J. R. Perilla, and S. Chandrasekaran, "Accelerating prediction of chemical shift of protein structures on GPUS: Using openACC," *PLoS Comput. Biol*., vol. 16, no. 5, May 2020, Art. no. e1007877, doi: 10.1371/journal.pcbi.1007877.
11. F. Milde, G. Tauriello, H. Haberkern, and P. Koumoutsakos, "SEM++: A particle model of cellular growth, signaling and migration," *Comput. Part. Mech*., vol. 1, no. 2, pp. 211–217, May 2014, doi: 10.1098/rspa.2021.0246.
12. D. Auddya *et al*., "Biomembranes undergo complex, non-axisymmetric deformations governed by Kirchhoff-love kinematics and revealed by a three dimensional computational framework," Jan. 2021, doi: 10.1101/2021.01.28.428578v1.

MATT STACK is a solutions architect at NVIDIA Corporation. His research interests include high-performance computing and performance analysis tools. Stack received his B.S. degree in computer science from the University of Delaware. Contact him at mstack@nvidia.com.

PAUL MACKLIN is an associate professor in the Department of Intelligent Systems Engineering at Indiana University. His research interests include open source computational tools for multicellular systems biology. Contact him at macklinp@iu.edu.

ROBERT SEARLES is a solutions architect at NVIDIA Corporation. His research interests include high-level programming models targeting heterogeneous systems, compilers, and performance optimization of high-performance computing systems. Searles received his Ph.D. degree from the University of Delaware. Contact him at rsearles@nvidia.com.

SUNITA CHANDRASEKARAN is an associate professor in the Department of Computer and Information Sciences, University of Delaware, and a computational scientist with the Brookhaven National Lab. Her research interests include high-performance computing, machine learning, and enabling interdisciplinary science by applying computer science concepts to real-world applications. Chandrasekaran received her Ph.D. degree in computer science engineering from Nanyang Technological University, Singapore. She is a Member of IEEE. Contact her at schandra@udel.edu.