GPU Acceleration of 3D Agent-Based Biological Simulations

Ahmad Hesam
ABS group
Delft University of Technology
Delft, Netherlands
a.s.hesam@tudelft.nl

Lukas Breitwieser
CERN openlab
CERN
Geneva, Switzerland
lukas.breitwieser@cern.ch

Fons Rademakers
CERN openlab
CERN
Geneva, Switzerland
fons.rademakers@cern.ch

Zaid Al-Ars
ABS group
Delft University of Technology
Delft, Netherlands
z.al-ars@tudelft.nl

***Abstract*—Researchers in biology are faced with the tough challenge of developing high-performance computer simulations of their increasingly complex agent-based models. BioDynaMo is an open-source agent-based simulation platform that aims to alleviate researchers from the intricacies that go into the development of high-performance computing. Through a high-level interface, researchers can implement their models on top of BioDynaMo's multi-threaded core execution engine to rapidly develop simulations that effectively utilize parallel computing hardware. In biological agent-based modeling, the type of operations that are typically the most compute-intensive are those that involve agents interacting with their local neighborhood. In this work, we investigate the currently implemented method of handling neighborhood interactions of cellular agents in BioDynaMo, and ways to improve the performance to enable large-scale and complex simulations. We propose to replace the kd-tree implementation to find and iterate over the neighborhood of each agent with a uniform grid method that allows us to take advantage of the massively parallel architecture of graphics processing units (GPUs). We implement the uniform grid method in both CUDA and OpenCL to address GPUs from all major vendors and evaluate several techniques to further improve the performance. Furthermore, we analyze the performance of our implementations for models with a varying density of neighboring agents. As a result, the performance of the mechanical interactions method improved by up to two orders of magnitude in comparison to the multithreaded baseline version. The implementations are open-source and publicly available on Github.**

***Index Terms*—agent-based modeling, simulation, GPU, co-processing, biological models, acceleration**

I. Introduction

Agent-based simulation (ABS) is a powerful tool for conducting research on complex biological systems. In ABS, a biological system is composed of a number of agents that individually are modeled to follow a fixed set of, often simple, rules. Agents can interact with neighboring agents or respond to external stimuli. Although the individual behavior of agents is often trivial, the emerging behavior that comes forth from the biological system as a whole can give researchers valuable insights [1]–[3].

As the complexity and scale of biological agent-based models increases so does the demand for computational power and efficiency [2]. Agent-based simulations are inherently parallelizable in their execution, as the agents' states can be modified independently of each other. Modern-day hardware is becoming increasingly more parallelized as a result of Dennard scaling [4] and the stagnation of Moore's law [5], as pointed out in [6]. Moreover, general-purpose computing on graphics processing units (GPUs) is an attractive solution to improve the computational efficiency of ABS applications in particular [7], [8], and parallel applications in general [9], [10]. By porting applications to, either fully or partially, run on GPUs it is possible to observe speedups of several orders of magnitude in comparison to the CPU-only execution [11]. Although several ABS frameworks exist that achieve significant speedups using GPUs in the field of ABS, there is still significant room for improvement, which we wish to address in this article.

BioDynaMo [6] is an open-source software platform for life scientists for simulating biological agent-based models. Each agent in BioDynamo is programmed to follow a specified set of rules, imposed by the modeler, that can trigger specified actions affecting itself or other agents. Agents in biological systems often interact with their local environment, and their behavior can be influenced by other agents that reside within a certain range. An example is the mechanical interactions a cellular agent undergoes when it physically collides with another agent. Local interactions are an extremely important concept in biological systems since it is the driving force behind key biological processes, such as tissue development [12].

BioDynaMo is fully parallelized using OpenMP and its performance scales with the number of CPU cores available on a system [6]. To further enhance the simulations' performance, we want to investigate the applicability of GPUs in accelerating compute-intensive operations in BioDynaMo. In this work we present the following contributions:

- Redesign the neighborhood search in BioDynaMo from a kd-tree method to a uniform grid method to profit from the parallel architecture of GPUs.
- Port the uniform grid implementation to GPU code using OpenCL and CUDA to address all major GPU vendors.
- Improve the GPU kernels based on domain-specific aspects of biological agent-based models.
- Benchmark the runtime and analyze the performance gains that are obtained.

The organization of the paper is as follows. Section II

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00040

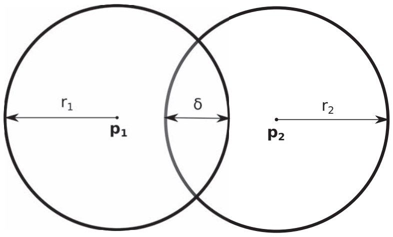


Fig. 1: Sphere-sphere collision force diagram (projected as circles for simplicity).

discusses related work. In Section III we define the problem in more detail. In Section IV we describe the methodology of our approach. Section V describes the hardware and software setup. In Section VI we present the results. And finally, in Section VII we draw the conclusions of this work.

II. Related Work

There are several frameworks and software packages that make it possible to simulate agent-based models for biological systems. There are many more specialized software solutions, but these generally focus on one biological process, or a few closely related biological processes. Some of the more general ABS frameworks for biological systems (BioCellion [13], PhysiCell [14], Timothy [15], and Chaste [16]) focus, among other things, on computational efficiency, but do not support GPU acceleration. In this work, we demonstrate that GPU acceleration is possible for general-purpose agent-based platforms.

In the works of [7] and [8] the authors present cellular agent-based simulation (ABS) programs that run entirely on a GPU. The authors report speedups of several orders of magnitude over ABS frameworks that are only targeted for CPUs. Although the findings are impressive, the fact that the simulation runs entirely on the GPU has two major drawbacks. First, it puts a lot of pressure on minimizing memory consumption. As GPU memory is a non-expandable and limited resource, there is a limit to the complexity of the agents' state and the scale of the model. In this work, we offload the most compute-intensive operation to GPU, which requires only a subset of the agents' state data to be present on the GPU memory. Second, operations that are independent of the agents, such as extracellular substance diffusion, are integral to biological systems and are absent from these works. With BioDynaMo we can simulate the extracellular substance diffusion efficiently on a multi-core CPU, independently from the GPU operations [6].

III. Problem Definition

The mechanical interaction operation is one of the most compute-intensive operations in any cellular agent-based model. Each cell (i.e. agent) interacts with all other cells within a certain interaction radius. For cells that are physically in contact with each other, we need to compute the collision forces and the resulting displacement. In BioDynaMo cellular agents can be physically modeled as spherical objects. For the scope of this paper, we shall consider only sphere-sphere interactions, as illustrated in Fig. 1 (projected as circles). Equation (1) [17] shows the calculations involved in determining the mechanical force.

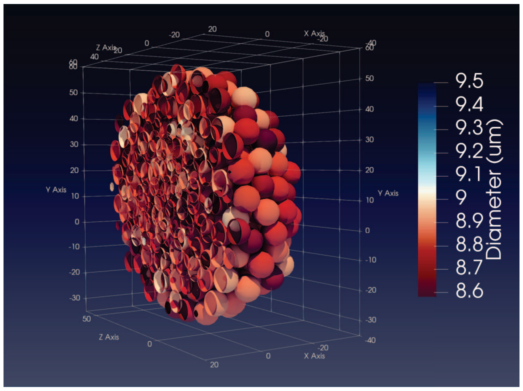


Fig. 2: A visualization of the cell division module in BioDynaMo (cross-sectional view). The colors represent the diameter of the cells.

$$
\begin{aligned}
\delta &= r_1 + r_2 - \|\mathbf{p_1} - \mathbf{p_2}\| \\
r &= \tfrac{r_1 \cdot r_2}{r_1 + r_2} \\
\mathbf{F} &= (\kappa \cdot \delta - \gamma \cdot \sqrt{r \cdot \delta}) \cdot \tfrac{\mathbf{p_1} - \mathbf{p_2}}{\|\mathbf{p_1} - \mathbf{p_2}\|} ,
\end{aligned} \qquad (1)
$$

where r_1 and r_2 are the radii of the spheres, $\mathbf{p_1}$ and $\mathbf{p_2}$ their position vectors, κ the repulsion coefficient, γ the attraction coefficient, and $\mathbf{F}$ the resulting collision force vector. After the collision force has been computed, we determine whether it is strong enough to break the adherence of the cell in question. If that is the case, then we integrate over the collision force to compute the final displacement. The length of the final displacement vector is generally limited by an upper bound.

To quantify the impact of improving this operation for BioDynaMo, we run one of the available benchmarks that use all default operations (cell division module). In this benchmark, a 3D grid of 262,144 cells of the same volume are spawned and proliferate for 10 iterations. Once the cells are instantiated, in each iteration the same operations are executed: 1) cell proliferation, 2) neighborhood lookup, and 3) resolving the mechanical forces. A visualization of cell proliferation in BioDynaMo with fewer cells and a longer runtime is shown in Fig. 2. We profile this benchmark to get a better understanding of the computational bottlenecks in BioDynaMo.

From Fig. 3 we observe that the mechanical interactions operation (highlighted in blue) is the most time-consuming in the benchmark by a large margin. Since this operation requires iterating over all agents, and in turn over all of their neighboring agents, this observation matches our prior expectation. 51% of the benchmark's runtime is spent on the mechanical force calculations as described in (1), and 36% is spent on updating the neighborhood list of each agent. Updating the neighborhood is executed in two steps: 1)

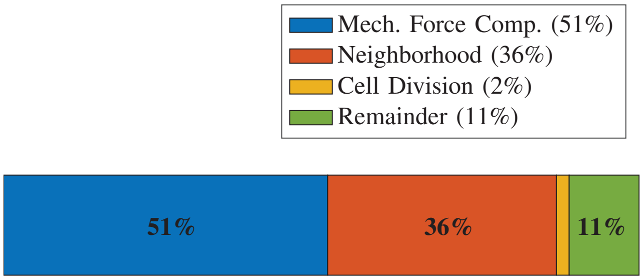


Fig. 3: Runtime profile of the cell division benchmark in BioDynaMo.

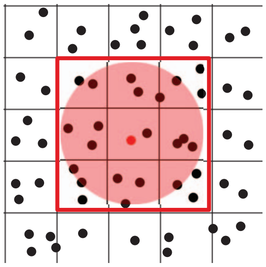


Fig. 4: Finding the neighborhood of an agent using the uniform grid method. Displayed in 2D for simplicity.

building a kd-tree, and 2) searching all the agents' neighbors within a specified radius.

A kd-tree is one of the many methods that can be used for a radial neighborhood search. Considering that we want to offload this mechanical interactions operation to GPU, a more appealing method could be a uniform grid method. The uniform grid method allows us to apply different techniques to improve the GPU version of the mechanical interactions operation, which we will discuss in this paper.

IV. Methodology

In this section we will go over the implementation of the various improvements that were made on the existing mechanical interactions operation in BioDynaMo. We use BioDynaMo v0.0.9-8b3d6c7 as the baseline version, which allows us to benefit more from GPU acceleration than the latest version presented in [6], as the data are stored in a structs-of-arrays format, rather than arrays-of-structs.

A. Uniform Grid Method

The uniform grid method imposes a regularly-spaced 3D grid within the simulation space. Each voxel of the grid contains only the agents that are confined within its subspace. Finding the neighboring agents of a particular agent can be done by only taking into account the voxels surrounding that particular agent, as illustrated in 2D in Fig. 4. The agent that we want to find the neighborhood for is colored red, and its interaction radius is highlighted in red. We only consider the agents in the 9 surrounding voxels (27 in 3D) around which a red line is drawn in the figure. We implement the uniform grid approach in BioDynaMo as a C++ class as illustrated in Fig. 5 as a UML diagram. For every simulation timestep, we reconstruct the uniform grid to take into account the addition, deletion, and movement of agents. Each voxel (i.e. `Box`) keeps track of the number of agents it contains and the last object that was added. Through the use of a linked list (`Grid::successors_`) we can iterate through all objects inside a single `Box`. The exact implementation details can be found in our Github repository[1].

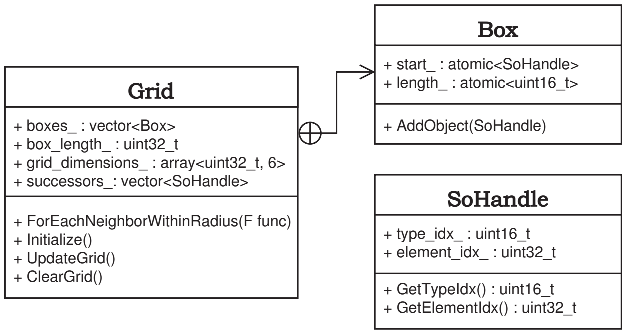

Grid
+ boxes_ : vector<Box>
+ box_length_ : uint32_t
+ grid_dimensions_ : array<uint32_t, 6>
+ successors_: vector<SoHandle>
+ ForEachNeighborWithinRadius(F func)
+ Initialize()
+ UpdateGrid()
+ ClearGrid()
Box
+ start_ : atomic<SoHandle>
+ length_ : atomic<uint16_t>
+ AddObject(SoHandle)
SoHandle
+ type_idx_ : uint16_t
+ element_idx_ : uint32_t
+ GetTypeIdx() : uint16_t
+ GetElementIdx() : uint32_t

Fig. 5: UML diagram of the class created for the uniform grid method.

B. GPU Implementation

We implement the uniform grid solution on the GPU using both CUDA and OpenCL to target GPUs from all major vendors. To minimize the amount of CPU and GPU context switches, we decided to port the uniform grid algorithm as well as the mechanical force computation as a single GPU kernel. Each GPU thread handles the mechanical interaction of one cell by 1) finding the cell's neighborhood, and 2) computing the mechanical forces between the cell and all the cells in its neighborhood. The state data of all the agents in BioDynaMo are stored as structs-of-arrays (e.g. the position data of all agents are store contiguously in memory). This allows us to copy the required state data for the mechanical interaction operation from the host DRAM to the GPU DRAM without first having to coalesce the data for all agents.

C. Improvement I: Reduction in Floating-Point Precision

BioDynaMo uses double-precision floating points (FP64) data types for all its floating-point data. However, most consumer GPUs perform stronger in single-precision floating-point (FP32) operations. This is a manifestation of the fact that GPU vendors primarily target the gaming industry and the field of artificial intelligence. Game engines and machine learning frameworks rely mostly on single-precision floating-point operations, so GPU manufacturers designed their consumer GPUs with more FP32 logic units than their double-precision counterparts. Some GPU vendors have dedicated cards for high-performance scientific computing that offer more FP64 logic units. For agent-based simulations, other factors, such

[1] https://github.com/Senui/biodynamo/tree/paper-floats

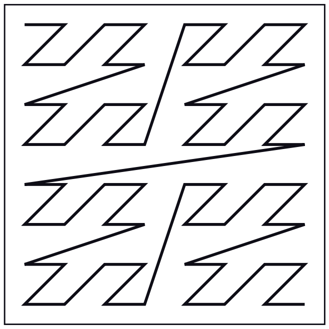
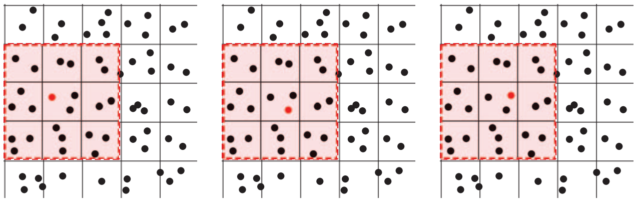


Fig. 6: The path of a Z-order curve in 2D. Adapted from [18].

as choosing the correct runtime parameters for a model (e.g. initial agent attribute values, number of simulation steps, etc.), generally far outweigh the accuracy of the final results in comparison to the imprecision that could come forth from reducing the floating-point precision from double to single. BioDynaMo has an extensive set of unit tests and integration tests that we can use to verify whether or not the reduction to FP32 affects the results. Moreover, FP32 data types are half the size of FP64 data types in memory, which reduces the size of the buffers that need to be copied back and forth from the host to the device, leading to a potentially significant increase in throughput, and thus performance.

D. Improvement II: Space-filling Curve Sorting

CUDA and OpenCL organize threads in groups of threads; called blocks and workgroups, respectively. The execution of the threads on the actual hardware is done in warps (generally in groups of 32 threads), with each warp executing the same instruction, but on different data (i.e. SIMT execution model). BioDynaMo lays down the agents' data in memory in the order that the C++ objects were instantiated. Each thread requires the data of the neighborhood of the simulation object it processes, which is not contiguous in memory, but rather scattered. Consequently, each thread performs numerous scattered memory accesses, which will in most cases end up fetching the data from DRAM, which can degrade the performance significantly. This could have been prevented if the data of agents that are close to each other in space are also laid down close to each other in memory. This is where space-filling curves come in; more specifically the Z-order curve [19]. A space-filling curve describes a path in multidimensional space that passes through the data points in consecutively local order, as illustrated in Fig. 6. A function that implements a space-filling curve can map multidimensional data (such as 3D Cartesian coordinates) to a one-dimensional array, where consecutive elements of that array are spatially local to each other. For a Z-order curve, the *Z-value* of each data point can be computed by binary interleaving its coordinate values and represents the index of the resulting one-dimensional array. With regards to BioDynaMo, this would imply calculating the Z-values of all the agents and sorting their state data accordingly. We anticipate that the cache line for accessing an agent will also contain the data of the agents in its neighborhood, and therefore reduces the number of fetches to DRAM. A reduced number of fetches to DRAM should lead to a less data-starved execution pipeline, and therefore a higher throughput, and thus a reduction in the execution time for each simulation step.



(a) Data required by thread X.

(b) Data required by thread Y.

(c) Data required by thread Z.

Fig. 7: Exploiting the reuse of neighboring simulation object data for the usage of shared memory resources on GPU.

E. Improvement III: Using Shared Memory

Most GPUs feature different types of on-chip memory, such as texture memory or shared memory. In certain cases, storing data on on-chip memory drastically reduces the latency for fetching data during a GPU kernel execution, and could therefore improve the overall performance. In BioDynaMo, the concept of letting each GPU thread handle the mechanical interactions of one agent leaves little room for the shared memory resources of GPUs to be used. The reason is that there is no reuse of data for threads within the same CUDA block (or OpenCL workgroup). The kernel parallelizes the for loop over all agents, so each thread works on data that are independent of the threads in the same block. To make use of shared memory, we need to create a kernel that allows multiple threads to work on mostly the same data. It is here where we can reap the benefits of the uniform grid method that we implemented as an alternative to the kd-tree method. We can exploit the fact that cells in the same voxel of the UG grid share the same neighboring voxels, and thus share the same simulation object candidates for their neighborhood. Instead of parallelizing the for loop over all cells, we consider a kernel that would parallelize a loop over all voxels. The threads that process the agents of a single voxel will need to reuse the neighborhood data, which can be stored in shared memory for low-latency memory fetches. The concept is illustrated in Fig. 7. All the state data belonging to the agents that are within the highlighted region in Fig. 7 are stored in shared memory. The shared memory objects are built in parallel by appending state data from agents of multiple voxels within the highlighted region. To avoid race conditions, the use of atomic operations is required in building the shared memory objects in parallel.

V. Experimental Setup

The hardware on which the evaluations are done belong to the CERN IT department and are tabulated in Table I. The CPUs of both systems consist of two physical sockets organized in a non-uniform memory access (NUMA) design.

TABLE I: Specifications of the systems used for benchmarking

	GPU chip	GPU RAM	Memory bandwidth	Single-precision performance	Double-precision performance	CPU chip	CPU cores	CPU DRAM
System A	Nvidia GTX1080 Ti	11GB	484 GB/s	11.34 TFLOPS	0.354 TFLOPS	Intel Xeon E5-2640 v4	20 (2 sockets, 40 threads)	256GB
System B	Nvidia Tesla V100	32GB	900 GB/s	15.7 TFLOPS	7.8 TFLOPS	Intel Xeon Gold 6130	32 (2 sockets, 64 threads)	187GB

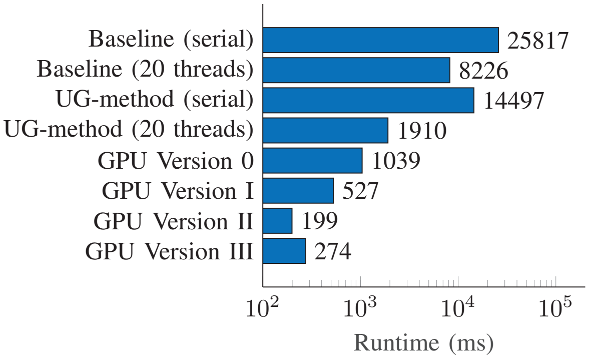


Fig. 8: The runtime for various implementations of the mechanical interaction operation running benchmark A. The GPU results are obtained from the CUDA runtime on system A.

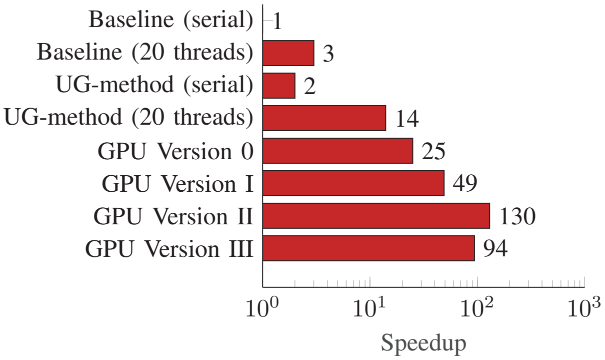


Fig. 9: The speedup with respect to the serial baseline version as obtained with benchmark A. The GPU results are obtained from the CUDA runtime on system A.

To mitigate cross-NUMA effects on some of the benchmark results, we run those benchmarks on only one socket of the NUMA domains. In practice, this was achieved by using the Linux utility tool `taskset`. In Section VI, we explicitly mention the benchmarks that were run on a single NUMA domain. The implementations and benchmarks can be found on Github[2].

To profile the GPU kernel and the performance metrics we made use of `nvprof`, which is part of the CUDA SDK Toolkit. Prior to recording the timing data for profiling GPU benchmarks, we run five iterations of the kernel to warm up the GPU. This measure is necessary for the following reasons: 1) the GPU could initially be in a power-saving state and therefore not perform optimally on the first run, 2) just-in-time compilation of the kernel requires more time on the first compilation, 3) additional time could be taken for transferring the kernel binary to GPU memory.

To quantify the performance of our solutions, we perform three types of analyses. First, we run the cell division benchmark (benchmark A) that was introduced in Section III. With this benchmark, we will quantify the performance of each solution in Section IV. Second, we created a benchmark (benchmark B) to analyze the performance among models with different local neighborhood densities. The cell division benchmark has a fixed average number of neighboring agents per agent, and therefore only represents models with the same neighborhood density. With the second benchmark, we vary the average neighborhood density by spawning two million agents on random positions in variable-sized simulation space. Consequently, the average number of neighboring agents per agent will be greater if the simulation space is smaller. To maintain a constant neighborhood density over the simulated time, we set the maximum displacement value of each agent to zero. The (neighboring) agents will stay locked in space, and therefore the neighborhood density will stay constant. The timing results of the benchmarks will exclude the model initialization time (creating the agents, assigning behaviors, etc.), and focus on the simulation performance. Thirdly, to understand the performance limitations of the current GPU implementation, we perform a roofline analysis [20] on the best performing GPU implementation. Through this analysis, we will understand how far the current implementation is from the maximum attainable performance on system B. We use the Empirical Roofline Tool (ERT) [21] to measure the empirical performance numbers of system B and to generate the roofline analysis plot. We retrieve the performance result (in GFLOP/s) and the arithmetic intensity (FLOPs/byte) of the GPU kernel with the use of `nvprof`.

VI. Results

Fig. 8 shows the runtimes obtained from running benchmark A for the various implementations of the mechanical interaction operation in BioDynaMo. Fig. 9 shows the obtained speedups comparison to the serial baseline version. Note that the x-axis is scaled logarithmically in both figures. The order of the bar charts follows from the order in which the versions were introduced in Section IV. Consecutive GPU versions include the implementation of the prior version, so for example

[2] https://github.com/Senui/hicomb_benchmarks

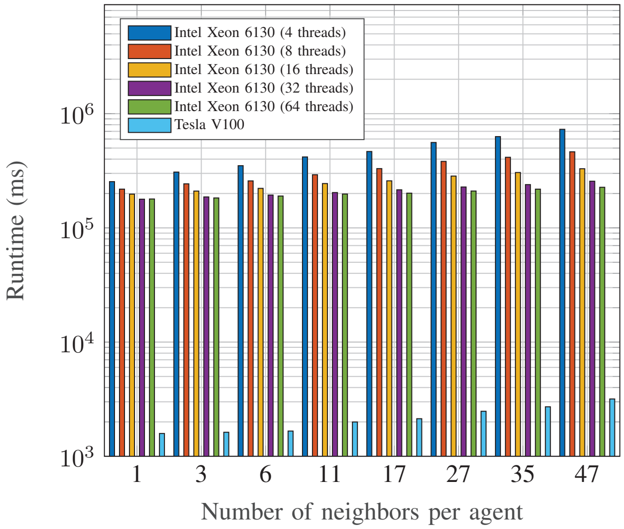


Fig. 10: The runtime of benchmark B for a varying neighborhood density. The Intel Xeon entries represent the baseline version. The Tesla V100 entries represent the best performing GPU implementation. The GPU results are obtained from the CUDA runtime on system B.

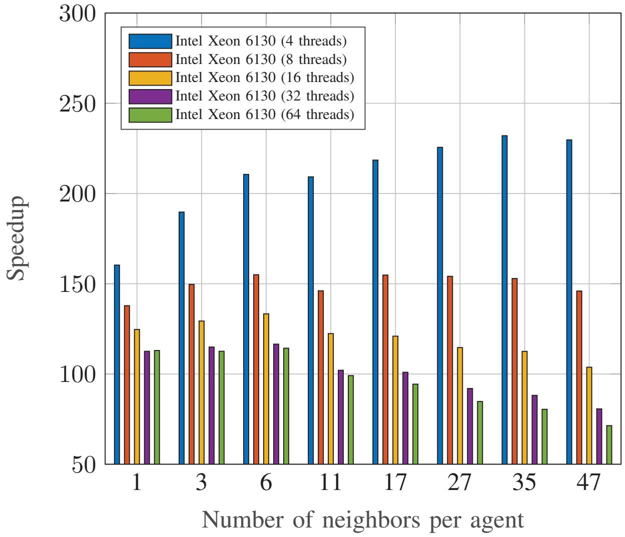


Fig. 11: The speedups with respect to the baseline version (for various numbers of threads) as obtained with benchmark B for a varying neighborhood density. The GPU results are obtained from the CUDA runtime on system B.

GPU version II includes the changes made for GPU version I. The results in Fig. 9 are obtained from running benchmark A on system A.

The serial uniform grid (UG) method performs twice as fast as the serial kd-tree method. On all 20 cores of the system (on a single NUMA domain), the UG method is $\frac{8226}{1910} = 4.3$ times faster than the kd-tree method. This can be attributed to the parallel construction of the uniform grid as opposed to the serial construction of the kd-tree.

The initial version of the GPU implementation (GPU version 0 in Fig. 9) of the UG method already offers an $\frac{8226}{1039} = 7.9\times$ speedup as compared to the multithreaded baseline version and is $\frac{1910}{1039} = 1.8$ times faster than its multithreaded CPU version. Even though the kernel is not yet optimized, due to the massively parallel architecture of the GPU we are able to attain a significant speedup compared to the multithread CPU version.

From Fig. 9 we can see about a $\frac{1039}{527} = 2.0$ speedup gained from reducing the data types that define a cell's state from doubles to floats. From Table I we can see that the FP32 throughput is 32 times greater than the FP64 throughput. From our speedup result, it becomes clear that the current GPU solution is limited by the memory bandwidth. Since FP32 data types are 4 bytes and FP64 data types are 8 bytes, the expected speedup of a GPU application that is memory bound and heavily relies on floating-point operations is two. We verified that the correctness of the simulations was not affected as a result of reducing the floating-point precision by running the unit tests and integration tests that are included in the testing suite of BioDynaMo.

Sorting the agents' state data based on a space-filling curve proved to reduce the execution time significantly, namely $\frac{527}{199} = 2.6$ times in comparison to the previous GPU version. This speedup confirms that the GPU kernel enjoys more spatial data locality when the agents' state data is sorted. As a result, memory accesses are more coalesced, which in turn leads to an increase in cache hits. This reduces the overall latency of obtaining the required neighborhood data from memory.

Redesigning the GPU kernel to utilize shared memory resources appears to worsen the overall performance by 28%. One of the reasons we found that causes the kernel performance to deteriorate, is the introduction of atomic operations in the kernel. The use of atomics is necessary to build the shared data structures that were introduced in Section IV-E in parallel. However, this causes stalling when multiple threads try to update the same shared data object. Moreover, the kernel needs to perform boundary checks on the blocks (CUDA) or workgroups (OpenCL) that are being executed by the GPU, which gives rise to thread divergence.

Fig. 10 and Fig. 11 summarize the results from running benchmark B on system B. The CPU results up to 32 threads were obtained by running on a single NUMA domain on system B. Fig. 10 shows the runtimes of the multithreaded baseline version (4, 8, 16, 32, and 64 threads) and the best performing GPU version (GPU version II), for a varying number of neighboring agents per agent. From the figure, it becomes clear that increasing the number of threads in a CPU-only runtime only reduces the runtime marginally, whereas GPU co-processing shows a significant reduction in runtime. Fig. 11 shows the speedup of the GPU runtime in comparison to the multithreaded baseline version. We observe that the speedup in comparison to the baseline version running with 4 threads lies between $160\times$ to $232\times$, depending on the neighborhood density. For the baseline version with 64 threads, the speedup lies between $71\times$ to $113\times$. These results imply that simulations that are densely populated, enjoy a speedup of up to two orders of magnitude when accelerating their workload with a GPU. Simulations that would normally take days on a multi-core CPU can be completed in hours on systems that feature a GPU. The significant reduction in simulation runtime allows researchers in the field of biological ABS to scale out their models and still obtain results rapidly.

In Fig. 11 we notice that the GPU performance gain stagnates, or even decreases, as the neighborhood density increases. The GPU kernel parallelizes the mechanical interaction computation for all agents, but the loop over all neighboring agents is serial. Consequently, this becomes the bottleneck for models with a high neighborhood density. We would like to investigate this solution by exploring *dynamic parallelism* [22] in existing GPU programming models. We hypothesize that parallelizing the serial loop over the neighborhood alleviates the bottleneck that is manifested in Fig. 11.

From the roofline model analysis in Fig. 12, we see that the best performing GPU implementation is still an order of magnitude away from the maximum attainable single-precision floating-point performance on system B. The data points are however close to the roof that represents the upper bound of the device memory bandwidth (HBM), which indicates that the kernel is close to being memory-bound. Future improvements to the kernel must focus on alleviating the strain on data transfer between the GPU and the GPU memory. Investigating other caching methods to bypass the HBM bandwidth roofline should be the main priority for future improvements. We observe that the kernel is able to attain higher performance with a higher neighborhood density. Based on the percentage of L2 cache reads relative to the number of total (L2 + HBM) memory reads, as obtained by `nvprof`, we believe this to be the result of increased cache reuse of the neighborhood state data per agent. For $n = 47$ this percentage is 41.3%, for $n = 27$ it is 40.6% and for $n = 6$ it is 39.4%.

VII. Conclusion

The goal of this work was to perform a comparative study of the acceleration potential of GPU co-processing in BioDynaMo to enable fast simulation of large-scale and complex biological models. To understand what the most effective way is to improve the performance of simulations such that large-scale and complex models can be implemented, we profiled the simulations that BioDynaMo is currently capable of running. We discovered that the mechanical interactions operation was the computational bottleneck by a large margin, due to the required data of local neighboring agents. We implemented a method alternative to the kd-tree method, the uniform grid (UG) method, which proved to be an excellent candidate for exploiting the parallel architecture of GPUs for performance gain. Not only did the UG method outperform the kd-tree method on CPU, but it opened up possibilities to exploit the advantages that GPUs offer. The final GPU kernel implementation resulted in speedups between $71\times$ to $232\times$ in comparison to the multithreaded baseline version, depending on the number of neighboring agents per agent and the number of threads the baseline is executed with. This result enables researchers of cellular agent-based models to rapidly obtain biologically insightful simulations with BioDynaMo.

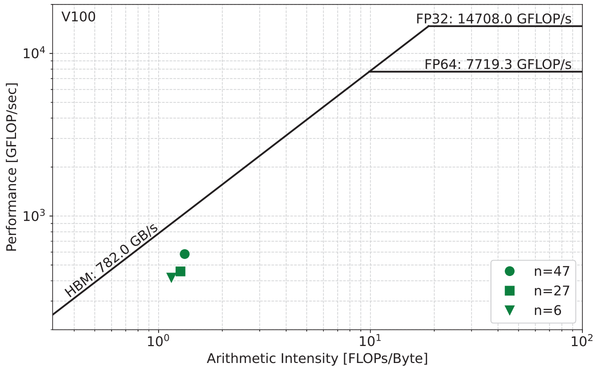


Fig. 12: GPU roofline model analysis of various neighborhood densities on system B, where n is the number of neighbors per agent.

Funding

This work was supported by the CERN Knowledge Transfer office [to L.B.] and CERN openlab [to A.H.].

References

[1] C. M. Macal and M. J. North, "Agent-based modeling and simulation," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, IEEE, 2009, pp. 86–98.

[2] G. An, Q. Mi, J. Dutta-Moscato, and Y. Vodovotz, "Agent-based models in translational systems biology," *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, vol. 1, no. 2, pp. 159–171, 2009.

[3] B. Di Ventura, C. Lemerle, K. Michalodimitrakis, and L. Serrano, "From in vivo to in silico biology and back," *Nature*, vol. 443, no. 7111, pp. 527–533, 2006.

[4] G. Fiori, F. Bonaccorso, G. Iannaccone, T. Palacios, D. Neumaier, A. Seabaugh, S. K. Banerjee, and L. Colombo, "Electronics based on two-dimensional materials," *Nature nanotechnology*, vol. 9, no. 10, pp. 768–779, 2014.

[5] G. E. Moore *et al.*, *Cramming more components onto integrated circuits*, 1965.

[6] L. Breitwieser, A. Hesam, J. de Montigny, V. Vavourakis, A. Iosif, J. Jennings, M. Kaiser, M. Manca, A. D. Meglio, Z. Al-Ars, F. Rademakers, O. Mutlu, and R. Bauer, *Biodynamo: A general platform for scalable agent-based simulation*, 2021. arXiv: 2006.06775 `[cs.CE]`.

[7] M. Lysenko and R. M. D'Souza, "A framework for megascale agent based model simulations on graphics processing units," *Journal of Artificial Societies and Social Simulation*, vol. 11, no. 4, p. 10, 2008, ISSN: 1460-7425. [Online]. Available: http://jasss.soc.surrey.ac.uk/11/4/10.html.

[8] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU," en, *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, May 2010, Publisher: Oxford Academic, ISSN: 1467-5463. DOI: 10.1093/bib/bbp073. [Online]. Available: https://academic.oup.com/bib/article/11/3/334/225993 (visited on 10/08/2020).

[9] S. Ren, K. Bertels, and Z. Al-Ars, "Efficient acceleration of the pair-hmms forward algorithm for gatk haplotypecaller on graphics processing units," *Evolutionary Bioinformatics*, vol. 14, p. 1 176 934 318 760 543, 2018, PMID: 29568218. DOI: 10.1177/1176934318760543. eprint: https://doi.org/10.1177/1176934318760543. [Online]. Available: https://doi.org/10.1177/1176934318760543.

[10] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. D. Zeeuw, and C. Strydis, "BrainFrame: A node-level heterogeneous accelerator platform for neuron simulations," *Journal of Neural Engineering*, vol. 14, no. 6, p. 066 008, Nov. 2017. DOI: 10.1088/1741-2552/aa7fc5. [Online]. Available: https://doi.org/10.1088/1741-2552/aa7fc5.

[11] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[12] P. Van Liedekerke, M. Palm, N. Jagiella, and D. Drasdo, "Simulating tissue mechanics with agent-based models: Concepts, perspectives and some novel results," *Computational particle mechanics*, vol. 2, no. 4, pp. 401–444, 2015.

[13] S. Kang, S. Kahan, J. McDermott, N. Flann, and I. Shmulevich, "Biocellion: Accelerating computer simulation of multicellular biological system models," *Bioinformatics*, vol. 30, no. 21, pp. 3101–3108, 2014.

[14] 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 computational biology*, vol. 14, no. 2, e1005991, 2018.

[15] M. Cytowski and Z. Szymanska, "Large-scale parallel simulations of 3d cell colony dynamics," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 86–95, 2014.

[16] G. R. Mirams, C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S.-J. Dunn, A. G. Fletcher, D. G. Harvey, *et al.*, "Chaste: An open source c++ library for computational physiology and biology," *PLoS computational biology*, vol. 9, no. 3, e1002970, 2013.

[17] A. Hauri, "Self-construction in the context of cortical growth: From one cell to a cortex to a programming paradigm for self-constructing systems," PhD thesis, ETH, 2013.

[18] Wikimedia Commons, *File:four-level z.svg — wikimedia commons, the free media repository*, [Online; accessed 20-August-2018], 2018.

[19] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," 1966.

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[21] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, *et al.*, "An empirical roofline methodology for quantitatively assessing performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, 2018, pp. 14–23.

[22] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, vol. 338, 2012, p. 2012.