Faster than the Speed of Life: Accelerating Developmental Biology Simulations with GPUs and FPGAs

A.S. Hesam

CE-MS-2018-24

Abstract

Life scientists are faced with the tough challenge of developing high-performance computer simulations of their increasingly complex models. BioDynaMo is an open-source biological simulation platform that aims to alleviate them from the intricacies that go into development. Life scientists are able to base their models on top of BioDynaMo's highly optimized core execution engine. At the core of all biological simulations is the mechanical interactions between possibly millions of objects. In this work we investigate the currently implemented method of handling mechanical interactions, and ways to improve the performance in order to enable large-scale and complex simulations. We propose to replace the existing kd-tree implementation for neighborhood operations with a uniform grid method that allows us to take advantage of architectures of hardware accelerators, such as GPUs and FPGAs. As a result, the multi-threaded uniform grid implementation accounts for a 14× speedup with respect to the serial baseline version. Accelerating the mechanical interactions through hardware acceleration proved to perform best on a GPU, with a resulting speedup of 134×.

Faster than the Speed of Life: Accelerating Developmental Biology Simulations with GPUs and FPGAs

by

A.S. Hesam

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Friday August 31, 2018 at 1:30 PM.

Supervisor:	Prof. dr. Z. Al-Ars	
Thesis committee:	Prof. dr. Z. Al-Ars,	TU Delft
	Dr. J. Sanders,	TU Delft
	Dr. F. Rademakers,	CERN openlab

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Dedicated to my parents,
for their endless love and encouragement

Contents

List of Figures

List of Tables

List of Acronyms

ABM Agent-Based Modeling

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specifc Integrated Circuit

BioDynaMo Biology Dynamics Modeler

CBM Continuum-Based Modeling

CUDA Compute Unified Device Architecture

CPU Central Processing Unit

Cx3D Cortex3D

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HPC High-Performance Computing

LUT Look-Up Table

OpenMP Open Multi-Processing

RAM Random-Access Memory

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SM Streaming Multiprocessor

SP Streaming Processor

UG Uniform Grid

Acknowledgements

First and foremost, I would like to thank my mentor, colleague and good friend, Lukas Breitwieser, who is a PhD student at CERN openlab and is the lead developer on the project I have worked on. From the many office discussions, pair programming sessions, to the coffee breaks and lunches, he really made my time in Geneva memorable, and I have learned a great deal from him.

Furthermore, I would like to thank my supervisor at CERN openlab, Dr. Fons Rademakers, for accepting me as one of his students. His vast knowledge in computing never seizes to amaze me. Also, his Dutch sense of humor never failed to amuse me.

I would like to thank Prof. Dr. Zaid Al-Ars, my supervisor at TU Delft, for his wisdom and guidance throughout my thesis. He made it my goal to impress him with this work.

My gratitude goes Dr. Roman Bauer, from the University of Newcastle, for the interesting discussions we had on the biological correctness of my efforts. I also appreciate the help of Aritz Iartza, as the system administrator of the CERN openlab (Techlab) computing resources; I appreciate the quick responses to my issues a lot. Many thanks to my friends at CERN and Delft for their emotional support.

Last, but certainly not least, my profound gratitude to my parents and sisters for their unfailing support and continuous encouragement. Without them I would not be where I am today.

A.S. Hesam
Delft, August 2018

1

Introduction

For as long as man can remember, he has tried to find answers to the seemingly unanswerable questions about life and the processes that make it up. Answers that, once found, could boost the quality of health far beyond current standards. Entire fields of science are dedicated to achieving this goal, combinedly called life sciences. Life scientists present theories and models that describe the processes that make up life and living organisms, ranging from molecular dynamics all the way to epidemiology. As the number of biological models is accumulating, man is slowly, but steadily learning more about the internal workings of living organisms. Not surprisingly, the time in which these models could be analytically verified is far in the past. Due to the sheer complexity and scale of these models, life scientists are bound to solve them through computer-driven simulations. This manuscript describes the contributions made to an emerging biological simulation platform with a specific focus on high-performance computing.

1.1. Motivation

Life scientists whose researches depend on simulations, but lack a strong background in IT, will find themselves in an awkward situation, as the days governed by Moore's law are long gone. There is a large gap between what the modeler *wants* to simulate, and what the modeler *can* simulate, and it is continuously increasing [4]. Implementations of biological simulations by life scientists tend to perform poorly and rely on outdated technologies. BioDynaMo is a project, hosted by CERN openlab, with the aim to alleviate life scientists from the burden of keeping up with the cutting-edge in computing technologies and their applications in life sciences. The project is based on a neuroscience simulation software Cortex3D [5] (Cx3D for short) that was originally developed at ETH Zürich. One of the goals of CERN openlab is to share the knowledge and expertise in IT that is present at CERN with communities beyond high-energy physics. To this extent CERN openlab formed a collaboration with Intel, Newcastle University, Kazan Federal University and Innopolis University in the effort to modernize the code from Cx3D, and generalize it for usage in more fields of life science (besides neuroscience). BioDynaMo stands for Biology Dynamics Modeler, and is an agent-based biological simulation platform that incorporates scalable and high-performance computing (HPC) technologies into its core engine. It offers life scientists the tools and functionality that they need to implement their model as a computer simulation, while BioDynaMo at its core takes care of the nitty gritty that is involved with high-performance computing. BioDynaMo is currently in a pre-release state, but is already being used in several researches, which allows for direct feedback from the life science community.

BioDynaMo is written in C++ and much effort has gone into parallel programming on central processing units (CPUs). These efforts alone have already improved the performance with respect to Cx3D significantly. HPC systems with hardware such as GPUs and FPGAs are becoming increasingly more programmable for general purpose computing. Moreover, the frameworks that support the use of these hardware accelerators in software development have strongly matured over the years. CERN openlab is currently in a phase of preparing for a new era in high-energy physics at CERN, called the High-Luminosity LHC (HL-LHC). The HL-LHC puts a huge strain on the IT infrastructure, with an estimated lack of computing power of two orders of magnitude. Hardware accelerators could potentially

close this gap of computing power, which is why CERN is putting efforts in research and development of the software that support the LHC experiments. This trend allow other sciences to reap the benefits that emerge from these efforts. This work is motivated by the exciting possibility that the advances in high-performance computing can open doors to novel research in many of the fields of life science. To the best of our knowledge there is currently no biological agent-based simulation software available for life scientists that makes use of hardware acceleration, and little effort has gone into the research thereof.

1.2. Thesis Scope and Contributions

Although BioDynaMo is already performing significantly better than its predecessor Cx3D, there are still computational bottlenecks that need to be alleviated. Several of the underlying simulation methods that are used in BioDynaMo can readily be mapped to fields other than life science, such as molecular dynamics, computational fluid dynamics, machine vision, and video game engines. The contributions in this work could therefore potentially be implemented or adapted to applications in these fields as well.

1.2.1. Thesis goal

The main goal of the thesis is formulated as follows:

> Perform a comparative study of the acceleration potential of GPUs and FPGAs of BioDynaMo's core to enable fast simulation of large-scale and complex biological models.

BioDynaMo has already been ported from Java (i.e. its predecessor Cx3D) to C++, to be closer to bare-metal and allow for the use of the programming frameworks that enable GPU and FPGA programming, which are mostly written in C / C++. In order to fulfill the above-stated goal, we need to execute the following work plan:

1. Profile simulations that make use of all the core modules in BioDynaMo to discover the computational bottleneck(s).
2. Identify and understand the methods and computations that are involved in the bottleneck.
3. Investigate alternative solutions to the already existing methods that could benefit significantly more from the use of hardware accelerators.
4. Port the code that implements the method or algorithm to be able to execute it hardware accelerators such as GPUs and FPGAs.
5. Improve the GPU and FPGA code with methods that are specific to the underlying architecture.
6. Benchmark the overall application, compare the results, and discuss the viability of a heterogeneous runtime of BioDynaMo.

1.2.2. Thesis contributions

The contributions of this thesis work include the following:

1. Identified the performance bottleneck in one of the core modules in BioDynaMo
2. Developed the extracellular diffusion module, which has been accepted in the upstream code base
3. Implemented an alternative algorithm to the radial neighborhood search
4. Ported alternative algorithm for execution on hardware accelerators (GPUs, FPGAs)

1.3. Thesis Outline

The structure of this manuscript strongly follows the work plan that has been laid out in Section 1.2.1. Chapter 2 will cover the theory that one should be familiar with to understand the presented work, including both the biological aspects and the high-performance computing aspects. In Chapter 3 we take a closer look at the inner workings of BioDynaMo, and profile some simulations in order to identify and understand BioDynaMo's computational bottleneck. We propose several alternative algorithms, which shall be compared to the current algorithm that makes up the bottleneck for potential replacement. The algorithm that will be deemed to be most feasible to be implemented on hardware accelerated systems, will be implemented in BioDynaMo, and the details can be found in Chapter 4. The evaluation thereof is presented in Chapter 5. The final conclusions, remarks and outlook for future works are given in Chapter 6.

2

Background

2.1. BioDynaMo

The Biology Dynamics Modeler, or *BioDynaMo* [6] for short, is an open-source project [7] that aims to create a general platform for simulating biological tissue dynamics in 3D space. Life scientists can use this platform to implement their biological models. At the core of BioDynaMo, several key features that can be used in simulations are implemented and highly optimized. This allows life scientists to directly use these features within their own model definitions, and alleviates them from the efforts that go into high-performance computing. As a result, BioDynaMo enables accelerated research in many of the life sciences.

The predecessor of BioDynaMo was a project named Cortex3D [8], or Cx3D for short, and was originally developed at ETH, Zürich. Cx3D was mainly focusing on the formation of neuronal network in the cerebral cortex. It was written in the programming language Java, which allowed life scientists to easily express their models, but at the cost of performance. BioDynaMo reuses the core concepts in simulating biological tissue dynamics of Cx3D, but with use cases not exclusive to neuroscience, but also for example immunology (e.g. cancer immunology). BioDynaMo is written in C++ to be able to be as close to bare metal as possible for high performance simulations.

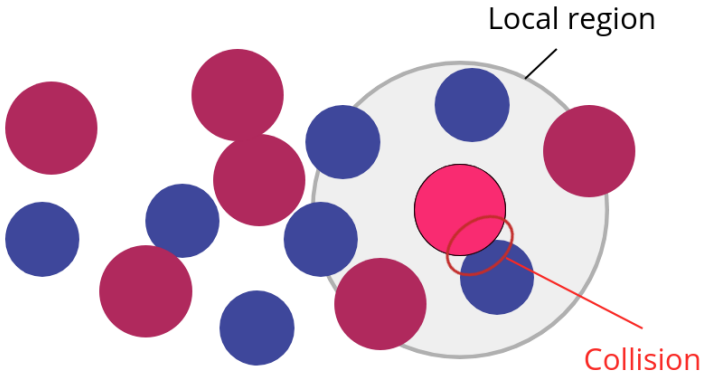


Figure 2.1: Agent-based modeling concept

2.1.1. Agent-Based Modeling

BioDynaMo is a software platform for life scientists for simulating biological *agent-based models*. In agent-based modeling (ABM) the objects that play a role in a model are expressed as individual autonomous entities, or *agents*. Figure 2.1 shows two types of agents, represented as blue and red circles. In molecular dynamics for instance, these agents would be atoms, while in cancer immunology these agents are generally cells. Each agent is programmed to follow a specified set of rules, imposed by the modeler, that can trigger certain actions or determine the interactions with other agents. Agents can be of the same scale, or of multiple different scales. The latter would make for a *multi-scale* agent-based

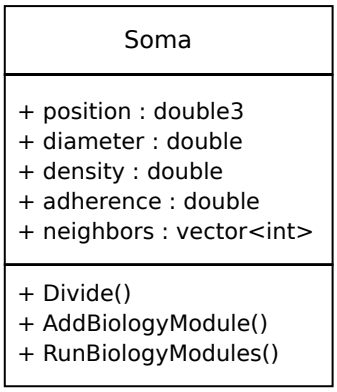


Figure 2.2: A simulation object in BioDynaMo

```
struct GrowthModule {
  double rate_;

  GrowthModule() : rate_(2) {}

  template <typename T>
  void Run(T* t) {
    t->SetDiameter(t->GetDiameter() * rate_);
  }
};
```

Listing 1: A biology module in BioDynaMo

model. In life sciences it is usually more interesting to consider multi-scale ABM, because organisms can often be decomposed into smaller organisms (e.g. human beings are essentially composed of trillions of cells). Therefore, BioDynaMo is not limited to simulating biology dynamics at a certain scale. To that extent, efforts are made, and still being made, to allow interaction with other software packages that handle higher or lower scale modeling, in addition to the possibility of extending BioDynaMo directly. Agents in BioDynaMo, such as cells, can interact with its local environment, and their behavior can be influenced by cells that reside within a certain range. An obvious example is the mechanical interactions a cell undergoes when it collides with another cell (see Figure 2.1). Local interactions are an extremely important concept in biology dynamics, since it is the driving force behind tissue development. For example, the formation of neural networks in the cerebral cortex is driven by the fact that neurons grow their axons towards other neurons who secrete chemical substances to let know of their presence. This process is known as axon guidance, and is still actively being researched. This kind of research would greatly benefit from a high-performance simulation platform, such as BioDynaMo, by allowing neuroscientists test their hypothesis through simulation.

2.1.2. Core Concepts

As already mentioned, BioDynaMo reuses most of the core concepts of Cx3D, which we shall briefly go over in this section.

Simulation Objects

In BioDynaMo an agent is referred to as a simulation object, and is implemented as a C++ object. Simulation objects are discrete physical components with a shape inherent to its type. For example, in neural development, a neuron can be modeled as a composition of two types of simulation objects: a sphere (somata) and chains of cylinders (neurites). Each simulation object can have several data attributes assigned to it, such as position, diameter, density, etc. Figure 2.2 shows the C++ class diagram of one way to model a soma cell.

Biology Modules

In BioDynaMo biology modules describe the biological behavior of a simulation object, and are implemented as C++ functors. A modeler defines biology modules and assigns them to the simulation objects. Listing 1 for example is a simple biology module that increases the diameter (i.e. the volume) of a simulation object of type `T`. If we wanted to assign this growth behavior to for example a soma cell, we would do `Soma::AddBiologyModule(GrowthModule)`.

Extracellular Substances

In cell biology or molecular biology the notion of extracellular substances refers to the substances that are physically outside of a cell. As described by Fick's law of diffusion these substances diffuse throughout the available space they are in. It is often through extracellular diffusion that cells are able to communicate with each other, which is why it is such an important phenomenon in biology dynamics. At the start of this thesis work the extracellular substances and the diffusion thereof was not implemented, and became part of this work, in order to enable more realistic simulations. Details on the implementation can be found in the next chapter.

2.1.3. Template Metaprogramming and Code Generation

The flexibility that BioDynaMo offers users, in terms of defining their custom simulation objects, biology modules, and extracellular substances, translates programmatically to heavy usage of template metaprogramming and code generation through preprocessor macros. The preprocessor and compiler can infer from user-defined compile-time parameters, what the internal structure of the data will be, and of what data type simulation states are composed. Without this design, the user would be responsible for registering all the customized objects and modules manually to a global resource managing class, which would significantly increase the complexity of developing simulations.

2.2. High-Performance Computing

For more than two decades Central Processing Units (CPUs) were the main driving force for increasingly faster and cheaper computer applications. Software was being developed under the assumption that with each new generation of CPUs the software would run faster. This assumption derived from the observation made by Intel's co-founder Gordon Moore in 1965 that every eighteen months the number of transistors per integrated circuit would double (i.e. Moore's law) [9]. However, since 2003 heat dissipation started to become a serious issue as the chips were getting increasingly denser packed with transistors, and clocked at higher frequencies. Processor vendors worked their way around this issue by manufacturing CPUs with multiple processing units (i.e. multi-core CPUs). This change in CPU architecture forced software developers to change their mindset on how to write code that will run faster on each new generation of CPUs. Sequential programs only run on one core of a multi-core CPU, which will not become any faster over the years due to aforementioned reasons. Software developers nowadays need to be familiar with the concept of parallel programming in order to enjoy the performance benefits that multi-core CPUs offer. The maximal attainable speedup of a parallel program is described by Amdahl's law in Equation (2.1):

$$S = \frac{1}{f + (1 - f)/N}, \tag{2.1}$$

where S is the maximal attainable speedup, f is the sequential fraction of the program, and N is the number of processing cores. Therefore, theoretically the maximum speedup could be achieved if the sequential fraction of the code is zero, and thus $S = N$. In other words, the performance of a parallel program scales with the number of cores. Programming interfaces like OpenMP allow developers to express where in their code they want to enjoy task-level parallelism (TLP), which modern compilers can execute on multi-core CPUs.

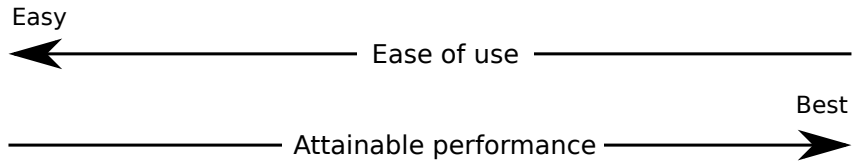
2.2.1. Graphics Processing Unit

A Graphics Processing Unit (GPU) is a computer chip that is specialized for rapid and efficient processing of large amounts graphics data. The computer architecture of a GPU is tailored for this purpose, and therefore has many processing cores that execute workloads in a massively parallel manner. For as long as GPUs exists, their parallel-processing architecture stayed consistent, but starting from around 2007 their processing cores became increasingly more programmable [10]. This development lead to the notion of general-purpose GPU programming, in which GPUs are programmed to perform computations other than graphics processing. Nowadays GPUs are used in fields such as machine learning, physics, finance, cryptography, and many more. The main reason for its success in each of these fields is because a lot of compute-intensive tasks can be processed in parallel. Even solutions to problems that are not parallelizable by nature can often be reformulated in another way that is parallelizable, and can benefit from execution on GPUs. From Equation (2.1) it becomes clear that the high number of processing cores, that characterize a GPU, leads to high performance computing capabilities. Moreover, in a lot of cases it is financially more appealing to purchase off-the-shelf GPUs, than a custom made CPU cluster, for solving a computational problem with about the same performance. The variety of ways to program GPUs also contributed significantly to the popularity of using GPUs as general-purpose hardware.

Programming

There are several programming frameworks available to program a GPU. For developers there is usually a trade-off between the ease of use of a framework and flexibility that the framework can offer to

tweak an application for performance. This tradeoff needs to be taken into account when choosing a framework for GPU programming. Figure 2.3 gives an overview of the most popular GPU programming frameworks and how they relate to each other in terms of programmability and performance. If the application in question is already part of an existing library that offers GPU acceleration, it is most likely the best choice to make use of that library. A reason for choosing not to include a library into one's application might be that only a few of the functions of the library are required, and cannot (easily) be disentangled, resulting in unwanted large binary files. Another reason could be incompatibilities with licensing. For those cases, and in the absence of a suitable library, a developer might be interested in programming GPUs through compiler directives. OpenACC and OpenMP are the two most popular interfaces to offload computational tasks to GPUs. Through the use of *pragma* statements, a developer is able to indicate that a piece of code should be run on one or more GPUs. As a result major code changes are often not required, and enables developers with limited knowledge on GPU technologies to accelerate their workloads. A disadvantage of GPU programming through compiler directives is the limited flexibility of deciding what code to run on the GPU. Furthermore, when an application's performance is bound by the bandwidth between the host (often a CPU) and the GPU, it is not possible to instruct the compiler through directives to alleviate this. For those cases, there are low-level APIs, such as CUDA and OpenCL, that make it possible to be in full control of the application's execution flow and data flow. This freedom comes at a price of increased complexity of coding and requires a deep understanding of the GPU's hardware and architecture.



Libraries	**Compiler directives**	**Low-level APIs**
• NVIDA Libraries • cuBlas, cuSolver, etc • ESSL / PESSL	• OpenACC • OpenMP	• CUDA • OpenCL
+ Easy to implement + Well-tested and community support - Limited functionality	+ Modification of existing programs with directives - Requires knowledge on data structures	+ Achieves highest performance results - Most time intensive - Requires expertise

Figure 2.3: Trade-off between ease of use and the attainable performance of GPU programming frameworks

Architecture
Figure 2.4 shows the architectural overview of a typical GPU (in specific, the Fermi architecture of Nvidia). In the figure we see an array of so-called Streaming Multiprocessors (SMs). Each SM contains a number of processing cores called Streaming Processors (SPs), as shown in Figure 2.5. An SP typically contains simple arithmetic logic units (ALUs) (simple in comparison to CPU ALUs), such as FP units and INT units as can be seen from the figure. Nowadays, GPUs are equipped with SMs containing specialized SPs that are faster in for example double-precision floating point operations, trigonometric operations, or matrix manipulations in machine learning algorithms, indicated as SFU in Figure 2.5.. Furthermore, the SPs in an SM share memory resources, such as registers and shared memory.

2.2.2. Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) is an integrated circuit on which a matrix of configurable logic blocks are connected via programmable interconnects. The fact that the interconnects are programmable allows one to construct logic that is specific to solving a particular problem (i.e. in a particular field), hence the name *Field-Programmable* Gate Array. FPGAs were originally used to for embedded systems where the logic could vary over time, and designing a dedicated application-specific

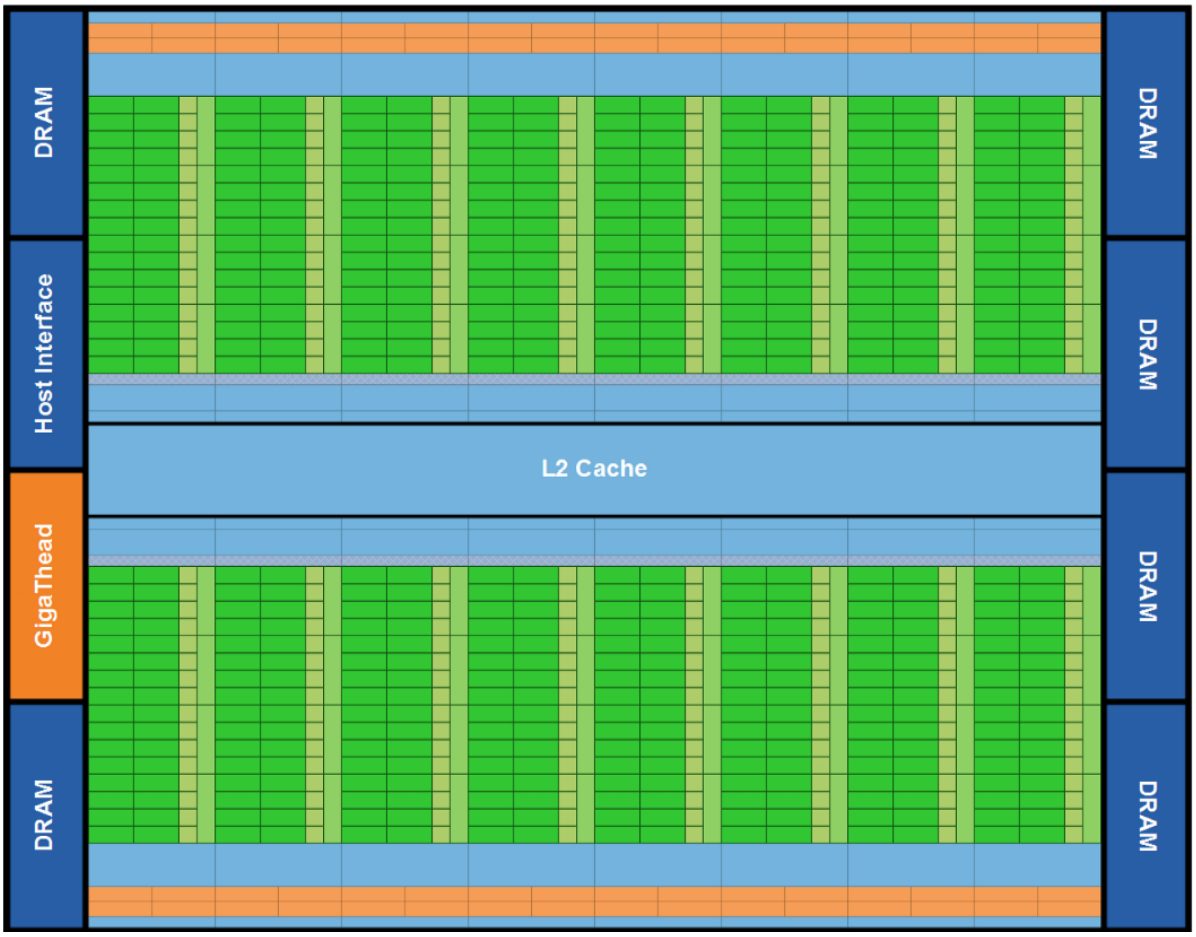


Figure 2.4: Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache) [1].

integrated circuit (ASIC) would be too expensive. Another use case of FPGAs was, and still is, to validate a circuit design before producing ASICs, in order to save costs and time.

Similar to GPUs, FPGAs have found their way into the field of HPC as hardware accelerators. HPC applications often have complex computations that need to be performed, that could take a CPU many clock cycles to complete due to the hardware being fixed and general-purpose. FPGAs on the other hand can be reconfigured to create the hardware that can do those complex computations in only a few clock cycles. Increasingly smaller reconfiguration times make it possible to reconfigure an FPGA within the runtime of a single application, to perform different types of complex computations. Cloud providers started incorporating FPGAs in their data centers for use in HPC applications [11].

2.2.3. Programming

Traditionally, FPGAs are programmed by means of a hardware description language (HDL), one use to describe the digital logic that the FPGA should be reconfigured to, down to the gate level. VHDL and Verilog are two HDLs that are most widespread used. Programming in FPGAs using HDLs from the bottom up allows circuit designers to create logic that is exactly suited to their specifications. Another method to program FGPAs is through high-level synthesis (HLS). HLS is an automated design process that generates from a high-level function description (e.g. using languages such as C or SystemC), the corresponding HDL files that can be used to synthesize the required logic. A logic synthesis tool (such as from Xilinx or Intel) can be used to perform the synthesis, placing and routing, and integrate the generated FPGA bitstream in a host binary application. Similarly to GPU programming, OpenCL can also be used to program FPGAs. An OpenCL kernel is fed as the input to a logic synthesis tool, and the generated FPGA bitstream is used to reconfigure the FPGA. In that sense, the OpenCL API serves as

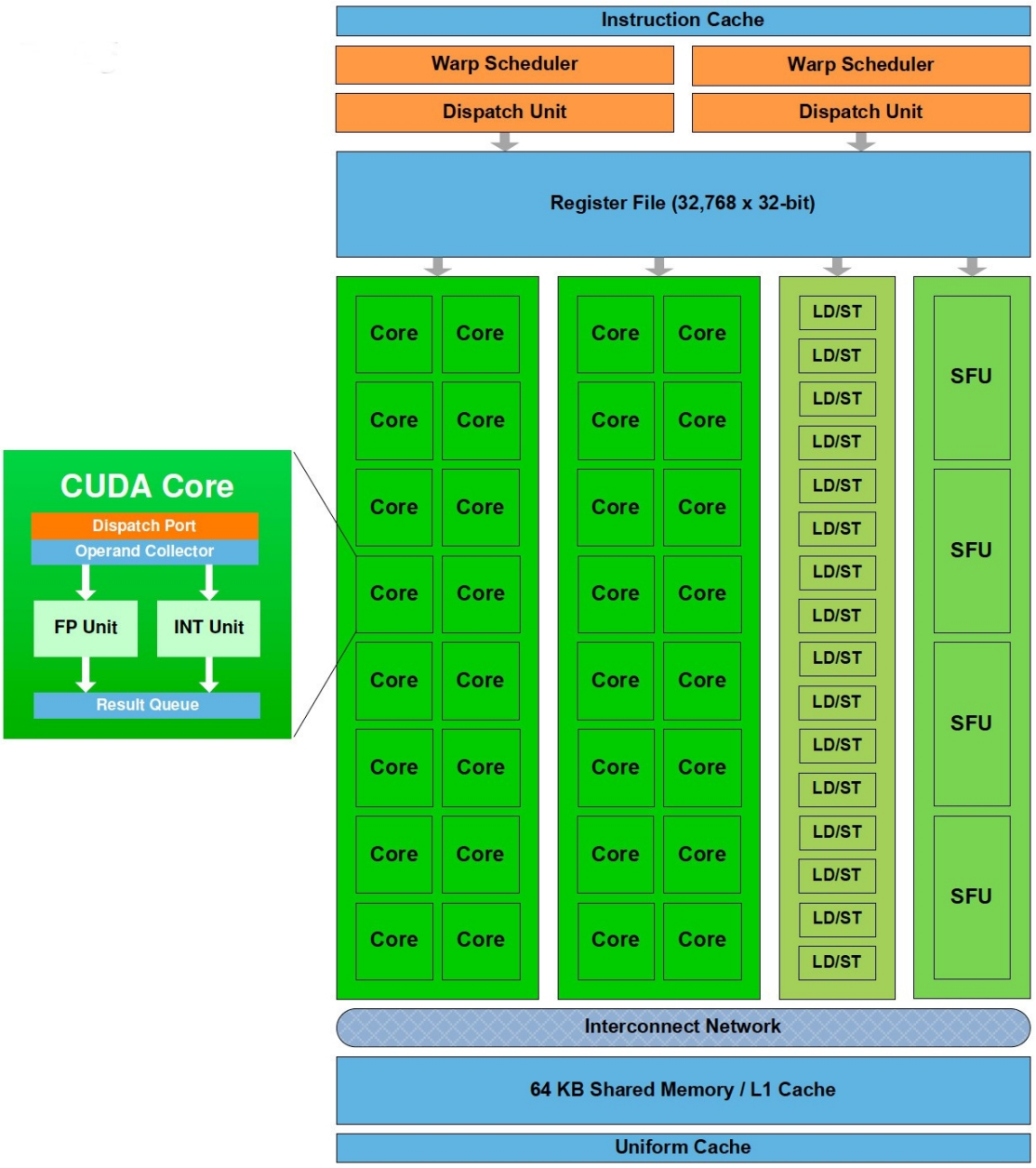


Figure 2.5: Fermi Streaming Multiprocessor (SM) [1].

an additional abstraction layer on top of HLS.

3

Algorithm

BioDynaMo currently has several features implemented that can be used for a wide range of biological simulations. In this chapter we will go over the execution flow of BioDynaMo simulations, and explain how the main features are implemented. The goal of this thesis is to find opportunities to improve the overall performance of BioDynaMo, and capitalize on them through the use of hardware accelerators. To that extent we need to determine which operations are the most compute intensive by profiling some typical simulations. Finally, we shall discuss our findings and suggest ways to improve the runtime.

3.1. Related Works

Agent-based simulations are one of the three methods that are used to study biology dynamics, with the other two being continuum-based modeling (CBM), and a hybrid of agent-based and continuum-based modeling. In CBM, a tissue or collection of organisms is treated as a continuum (i.e. a non-discrete object), and one is able to learn about the tissue's overall morphology and accumulated properties [12]. With CBM the influences of individual cells or organisms is neglected, and are abstracted by formulas that describe the global influence of a region of cells or organisms.

There exist several frameworks and software packages that make it possible to simulate biology dynamics. There are many more specialized software solutions, but these generally focus on one biological process, or a few closely related biological processes. One of the more general software solutions is Chaste [13], a multiscale computation framework for modeling cell populations. Chaste's implementation involves modeling the cellular behavior (i.e. cell division or cell death), modeling the mechanical interactions between cells, and the transport of signaling molecules between cells. The cells can be either positioned in a lattice-fixed organization or lattice-free.

In molecular dynamics (MD) simulations the goal is to simulate the physics that plays a role at an atomic scale. Some of the forces in MD simulations (e.g. due to the electrical potential between atoms) have an interaction radius that spans over the entire simulation space, and are $O(N^2)$ in time complexity when solving them numerically. These types of forces are generally the performance bottlenecks. In order to improve performance, these forces are approximated by only taking into account atoms within a certain *cutoff radius*. This becomes a type of problem similar to computing the physics in agent-based simulations (e.g. BioDynaMo). In the work of [14], the authors investigated the performance gains of deploying MD simulations on a GPU-accelerated supercomputer (TH-1A). Similar to our approach, the authors divided the space among the compute nodes, and offloaded the physics to the GPU that is on each node. The thread organization of the kernels is adapted to the number of voxels and particles the GPU needs to perform the computation on. As one of their optimizations, they employed the CUDA stream model [15], which effectively hides the memory transfer from and to the host and device, by overlapping the transfers with computations. Additionally, several optimizations were done to cache the particle data in the shared memory. As a result, an overall speedup of $24 - 26\times$ is achieved on a single node with 2.25M particles w.r.t. the serial CPU execution. Moreover, the authors show that executing the physics on both the CPU and GPU (hybrid) on 3000 nodes outperforms the pure CPU code on 6000 nodes, and the pure GPU code on 5000 nodes.

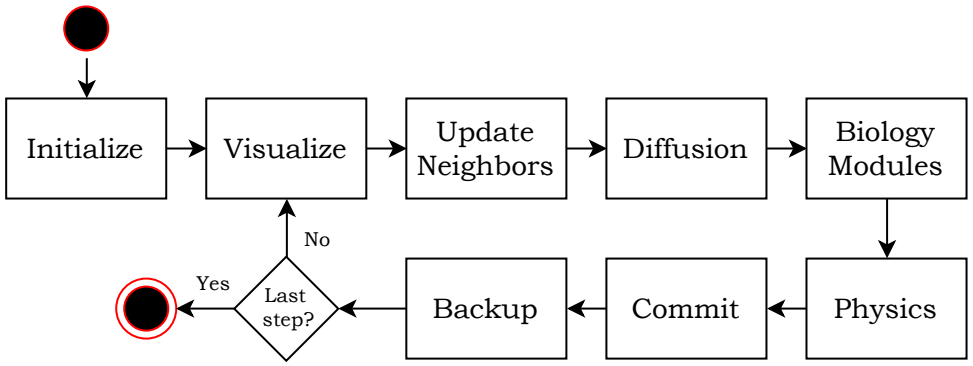


Figure 3.1: BioDynaMo's execution flow

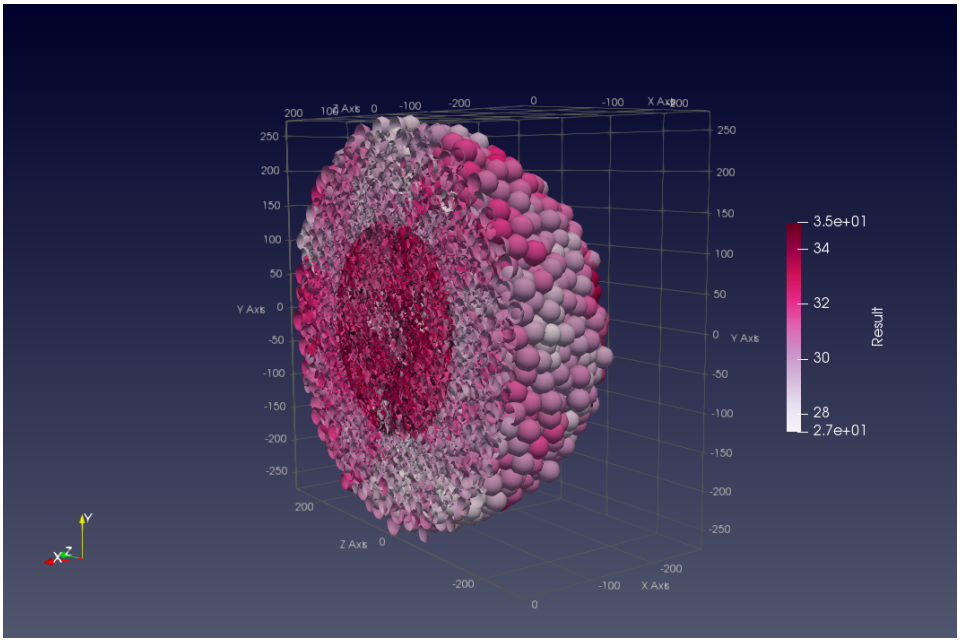


Figure 3.2: Visualization of a conceptual cancerous tumour with BioDynaMo

3.2. BioDynaMo Simulations

Figure 3.1 shows BioDynaMo's core execution flow. In this section we shall go into more detail about each of the steps in the execution flow.

Visualize

If enabled, this invocation will create the visualization objects for the current timestep. These objects can either be exported to file, or piped to the visualization software (ParaView) directly for live visualization. The visualization module was implemented as part of this thesis work. An example of the visualization can be seen in Figure 3.2.

Update Neighbors

Each simulation object maintains a list of its closest neighbors' identifiers, within a range defined by the modeler. Simulation objects can constantly move, and therefore the neighbors of an object change. Therefore we need to update the neighbor list of each simulation object every timestep. This requires an algorithm that can find the neighbors within a certain radius, and update each simulation object's neighbor list. Currently in BioDynaMo this is achieved by a *kd-tree*, which is a space-partitioning data structure.

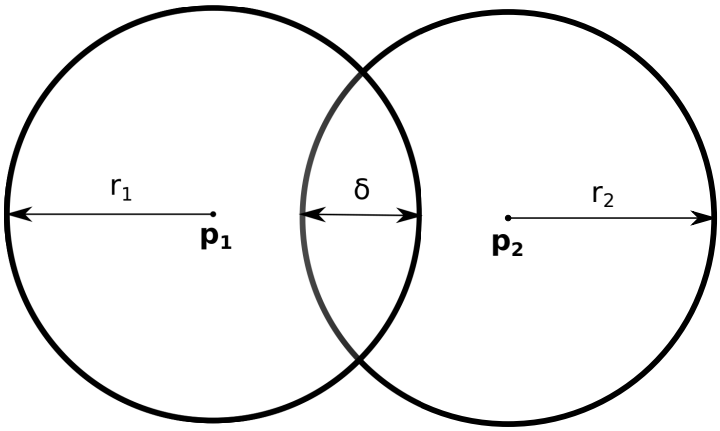


Figure 3.3: Sphere-sphere collision force diagram

Diffusion
As mentioned in Section 2.1.2 the extracellular diffusion applies Fick's law to the extracellular substances that are present in a simulation. These substances are of molecular scale, so they are modeled as concentration values in a 3D Cartesian grid. The partial differential equation that simulates diffusion is computationally solved with a central difference method that updates each grid point in the Cartesian grid as described in Equation (3.1):

$$\begin{aligned} u_{i,j,k}^{n+1} = u_{i,j,k}^{n} &+ \frac{\nu \Delta t}{\Delta x^2}(u_{i+1,j,k}^{n} - 2u_{i,j,k}^{n} + u_{i-1,j,k}^{n}) \\ &+ \frac{\nu \Delta t}{\Delta y^2}(u_{i,j+1,k}^{n} - 2u_{i,j,k}^{n} + u_{i,j-1,k}^{n}) \\ &+ \frac{\nu \Delta t}{\Delta z^2}(u_{i,j,k+1}^{n} - 2u_{i,j,k}^{n} + u_{i,j,k-1}^{n}), \end{aligned} \tag{3.1}$$

where $u_{i,j,k}^{n}$ is the concentration value on grid point (i, j, k) at timestep n, ν is the diffusion coefficient, Δt is the duration of one timestep, and Δx, Δy, and Δz are the distances between grid points in the x, y, and z direction, respectively.

Biology Modules
In this step we apply the biology modules on each cell. The computations that are involved in this operation depend on the model. An example of a biology module could be similar to the one in Listing 1. One could extend that biology module for instance with an additional rule that a cell divides into two cells when it reaches a certain threshold volume. This is a crude model that represents cell proliferation. Cell division is already implemented in BioDynaMo, and does not need to be implemented by the modeler.

Physics
For simulation objects that are physically in contact with each other we need to compute the collision forces and the resulting displacement. For the scope of this thesis we shall focus only on sphere-sphere interactions, as illustrated in Figure 3.3 (projected as circles). Equation (3.2) shows the calculations involved in determining the mechanical force.

$$\begin{aligned} \delta &= r_1 + r_2 - \|\mathbf{p_1} - \mathbf{p_2}\| \\ r &= \frac{r_1 \cdot r_2}{r_1 + r_2} \\ \mathbf{F} &= (\kappa \cdot \delta - \gamma \cdot \sqrt{r \cdot \delta}) \cdot \frac{\mathbf{p_1} - \mathbf{p_2}}{\|\mathbf{p_1} - \mathbf{p_2}\|}, \end{aligned} \tag{3.2}$$

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 simulation object

in question. If it is, then we integrate over the collision force to compute the final displacement (whose absolute value is generally limited by an upper bound).

Commit
The computations that are involved in RunBiologyModules or RunPhysics will most likely alter the state of the simulation objects. However, we cannot update their states directly after each computation, as this would lead to a situation where some simulation objects already advanced to the next state, while others are still in the current state. In reality, biological tissue develops in continuous time, with all the organisms changing their states simultaneously. Therefore, we keep track of the incremental changes that result from each state-altering computation, and update the attribute data in this Commit step. The duration of a timestep determines the resolution of the simulation, in terms of mimicking reality.

Backup
This is an optional step that enables BioDynaMo to make periodic backups of the entire simulation state. This is a useful feature for long-running simulations, where a technical failure of the machine could occur during the simulation. The state of the entire simulation is serialized to disk using ROOT's I/O module [16], and can be restored to memory upon request. For more in-depth information about this step, the reader is advised to study [17].

3.3. Profiling

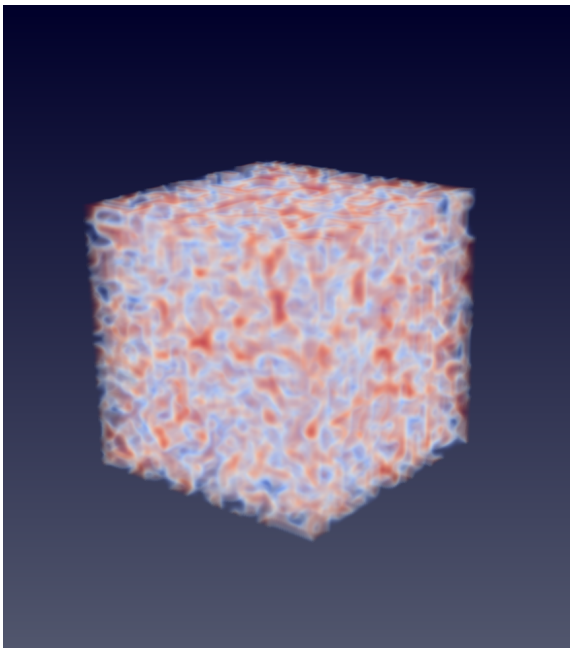
In order to determine and identify computational bottlenecks, it is necessary to profile typical simulations in BioDynaMo. For that purpose we created a simulation that simulates the early stages of brain development. In more concrete terms, the simulation consists of tens of thousands of neural progenitor cells that biochemically interact with each other in 3D space. By studying their behavior one could potentially identify treatments for neurodevelopmental diseases, such as epilepsy, autism and schizophrenia [18]. Each cell can proliferate, migrate, and secrete and detect extracellular substances. In this specific benchmark there are two type of progenitor cells. Each type secretes a different chemical substances. A cell is attracted by the substance it secretes, and moves according to the concentration gradient. As a result, cells of the same type are attracted to each other and form clusters of cells. Figure 3.4a and Figure 3.4c respectively show the beginning and end state of the extracellular substance secreted by one of the type of cells. Figure 3.4b and Figure 3.4d respectively show the beginning and end state of the progenitor cells. As expected cluster formations are established, with properties in accordance to biological theory (in terms of cluster density, cluster size, uniformity, etc.).

Figure 3.5 shows the distribution of the runtime of the benchmark. An important note to make here, is that the UpdateNeighbors step is included in the physics results, because it is the only module that requires knowledge of the local environment.

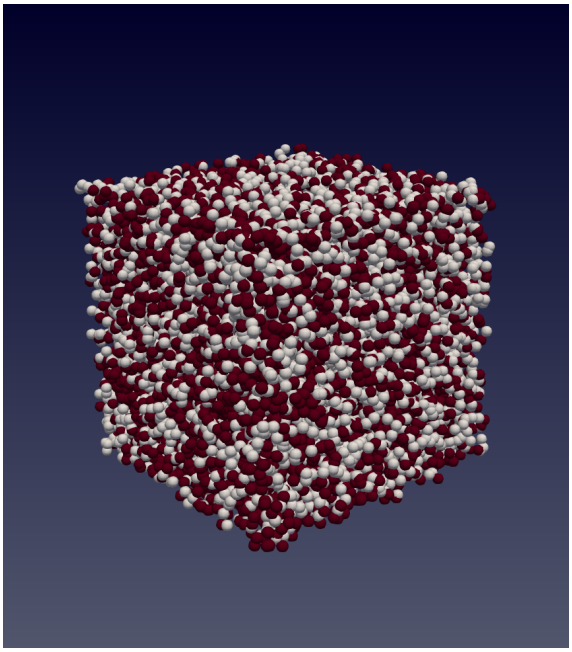
3.3.1. Insights

So from Figure 3.5 it becomes clear that the physics operation is by far the most time consuming. In the physics execution time, about 75% of the time is spent in UpdateNeighbors, and 25% in the operations involved with mechanical interactions. UpdateNeighbors is executed in two steps: 1) building the kd-tree, and 2) searching all the cells' neighbors. From profiling the UpdateNeighbors operation we found that about 30% of the time is spent on constructing the kd-tree, while 70% of the time is spent on the neighbor search. Depending on the use case with BioDynaMo, the kd-tree might not be right approach for the UpdateNeighbors operation. In biology dynamics, cell proliferation, apoptosis and migration are frequently occurring phenomena, which programmatically are implemented as allocations, deletions and position updates of the simulation objects. Consequently, the benefit of constructing a kd-tree once and performing neighbor searches multiple times is lost. We are required to reconstruct the kd-tree every timestep to deal with these dynamic scenes. Moreover, in the context of high-performance programming, it is not trivial to parallelize the operations in a kd-tree efficiently. Improvements in the overall execution time would most certainly include a faster way to search for the neighborhood of the simulation objects.

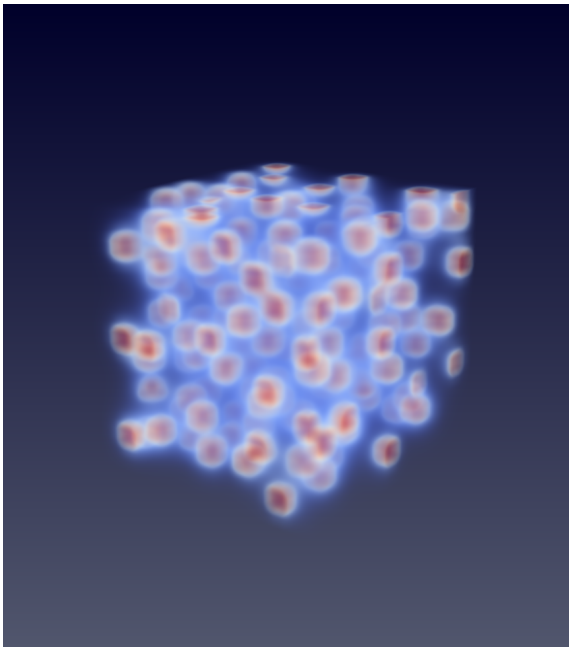
The execution time of the mechanical interactions is also a significant portion of the overall runtime of the benchmark simulation. Appendix A includes the pseudocode of the mechanical interaction computation. Compared to the other default computations in BioDynaMo, the computation of the me-



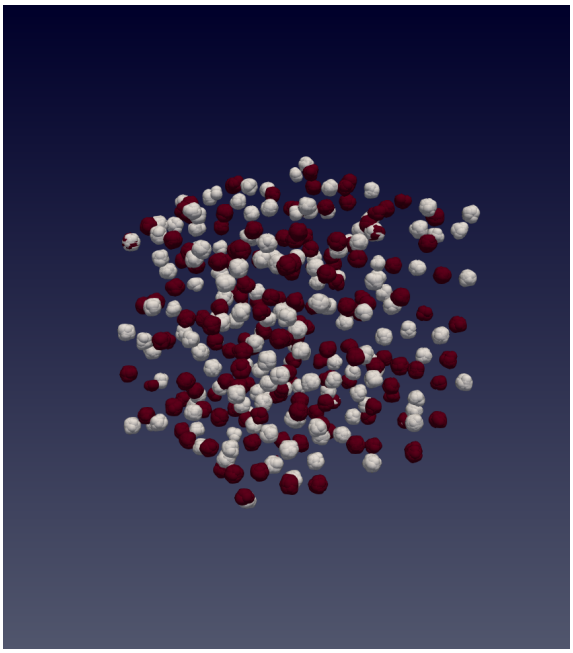
(a) Begin state, substance concentration



(b) Begin state, cells



(c) Final state, substance concentration



(d) Final state, cells

Figure 3.4: The visualization of chemotaxis. Cells move towards the gradient of their self-secreted substance type and form clusters as a result.

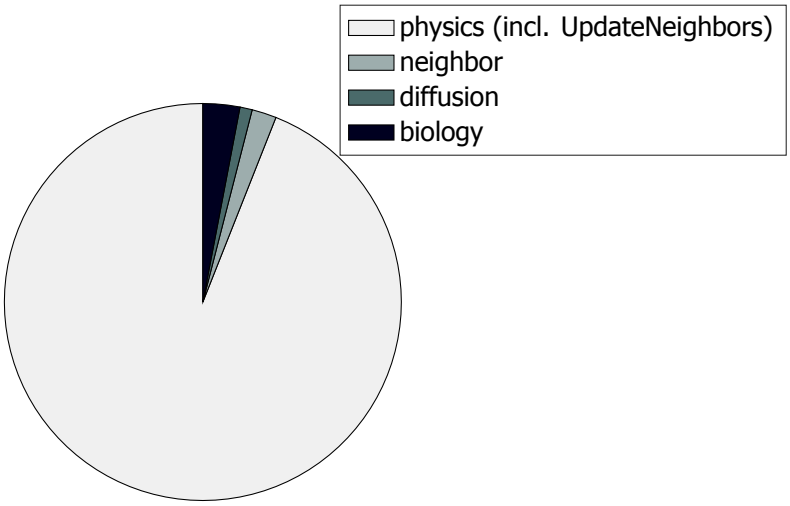


Figure 3.5: Percentage of execution time of each computation type.

chanical interactions is currently the only one that contains a doubly-nested for loop (over all cells, and over all neighbors of each cell).

3.3.2. Thoughts for Improvement

In order to improve the overall performance of BioDynaMo we must reduce the execution time of the neighborhood search that is required for the mechanical interactions. There are several avenues which upon investigation could lead to performance gain. Four of these shall be addresses in this work:

1. Algorithmic alternatives: investigate the use of an algorithm other than the kd-tree to find the neighborhood of simulation objects. A strong preference is given to algorithms that are strongly and trivially parallelizable to be able to boost the performance even further with the solutions that follow.
2. CPU acceleration: investigate ways to accelerate the selected algorithm on multi-core CPUs through a multithreaded execution.
3. GPU acceleration: investigate ways to make use of modern GPUs' massively parallel computing capabilities to accelerate the selected algorithm.
4. FPGA acceleration: investigate ways to make use of the reconfigurable design properties of FPGAs to accelerate the selected algorithm.

We cannot identify in advance which of these avenues are most effective for our purpose, because of the many degrees of freedom that each solutions has in terms of potential performance gain. For that reason we must implement all of them, compare the speedups that we obtain, and categorize the solutions based on the metrics that highlight their respective benefits the most.

3.4. Algorithmic Alternatives

3.4.1. Octree

The classic rival of a kd-tree solution is the octree. An octree divides the initial space with data points into eight equal subspaces, or *octants*. Each octant has the capacity to host only a specified number of data points. If the number of data points is higher than the specified number, the octant splits into eight other octants. This process is recursively repeated until the capacity condition of all the octants is met. Figure 3.6 is a visualization of the resulting data structure of the octree's 2D variant (quadtree).

There exist multiple ways to use this data structure in performing radial neighbor searches, but the crux of most of these methods is the possibility to prune large sections of empty space, as can be seen in Figure 3.6. To measure the performance of octrees against the kd-tree implementation, we integrated

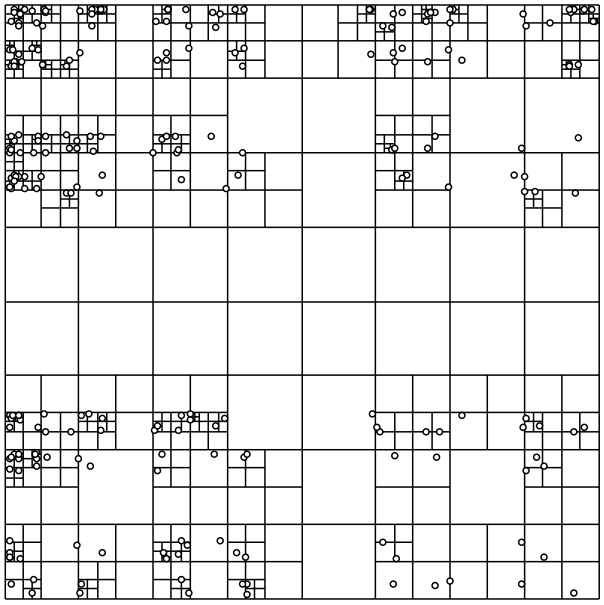


Figure 3.6: An example of a quadtree: the 2D variant of an octree [2].

three octree implementations in BioDynaMo and profiled the execution time of the UpdateNeighbors step like we did in Section 3.3 [1]. Figure 3.7 shows the performance of the octree implementations with respect to the current kd-tree implementation [2].

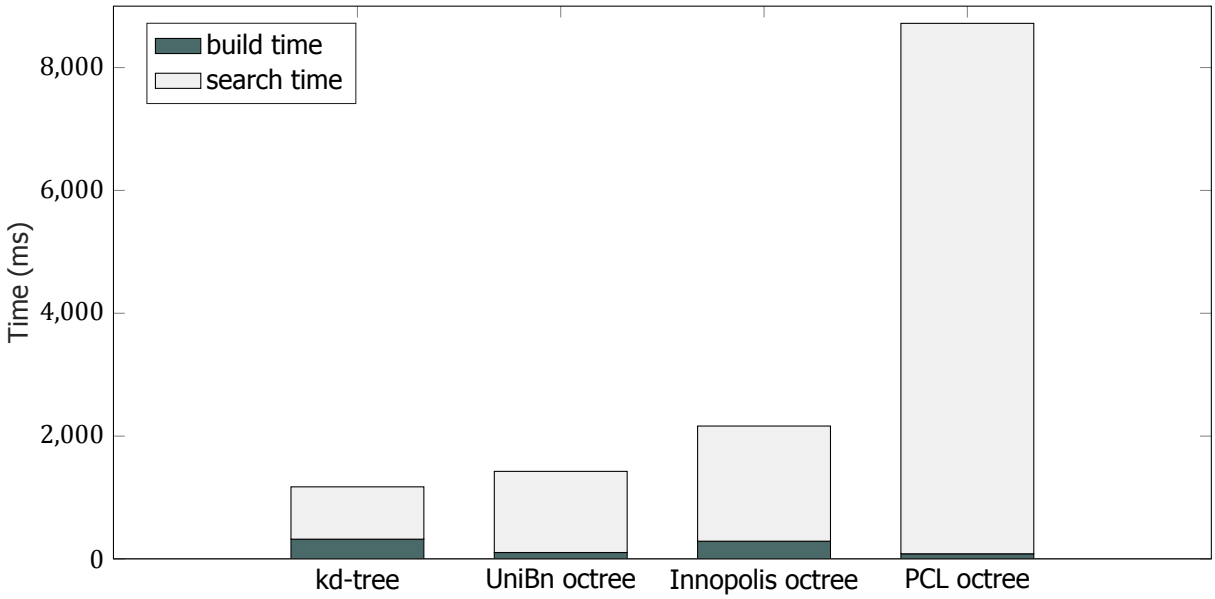


Figure 3.7: Preliminary benchmark results of octree method.

We observe that octrees do not perform better than the current kd-tree implementation. Therefore we exclude octrees as an alternative approach to perform radial neighborhood searches.

3.4.2. Uniform Grid Method

The uniform grid (UG) method is a more simplistic approach to achieve neighborhood searches. To understand the intuition behind the UG method it is good to think about the brute-force method of

[1] https://github.com/BioDynaMo/biodynamo/tree/octree-benchmark
[2] https://github.com/BioDynaMo/biodynamo/tree/thesis-bench-kdtree

radial neighborhood searching. The time complexity of the brute-force calculation is $O(N^2)$, where N is the number of cells. In such a method we compare the distance between each cell with all other cells, and decide whether it is smaller or equal to the interaction radius in order for a neighbor cell to be considered part of a neighborhood. The UG method imposes a regularly-spaced 3D grid within the simulation space. Each voxel of the grid contains only the cells that are confined within its subspace. Finding the neighboring cells of a particular cell can now be done by only taking into account the voxels surrounding that particular cell, as illustrated in 2D in Figure 3.8. The cell that we want to find the neighborhood for is colored red, and its interaction radius is highlighted in red. We only consider the cells in the 9 surrounding voxels (27 in 3D) around which a red line is drawn in the figure. The time complexity now becomes $O(kN)$, where k is the number of cells in the surrounding voxels.

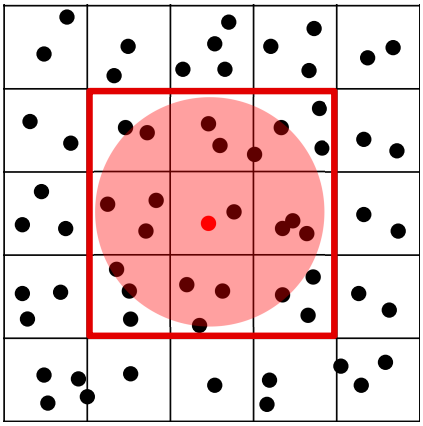


Figure 3.8: Finding the neighborhood of a cell using the uniform grid method.

Similarly to what we did with the octree approach, we implement the uniform grid method in BioDynaMo, and compare the performance results with the kd-tree implementation. We set the voxels' edge length equal to the largest object in the simulation to avoid the possibility that a simulation object can span over multiple voxels. Figure 3.9 shows the results.

As we can see from Figure 3.9 the UG method performs significantly better than the kd-tree method. Moreover, we anticipate the UG method to be more a more straightforward to parallelize, and thus a more effective method to exploit the benefit of hardware accelerators with. The UG method is also a more natural approach for a future distributed runtime of BioDynaMo. The simulation space can be divided among the nodes in a cluster or cloud through the concept of voxels that is inherent to the UG method. We therefore select the UG method as the alternative to the kd-tree method.

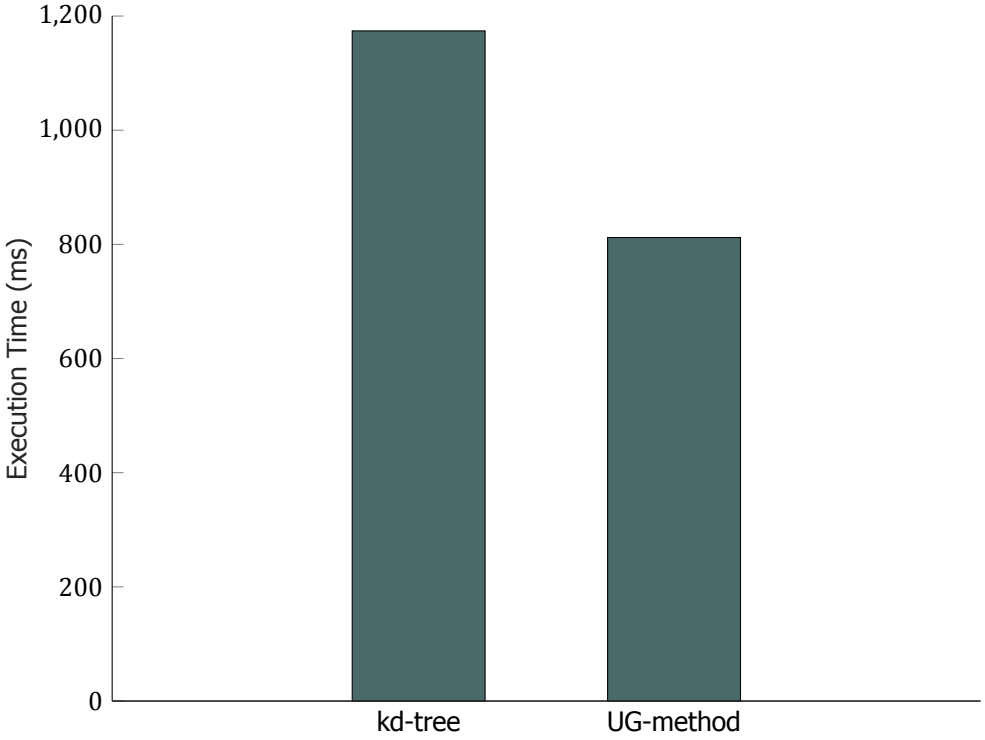


Figure 3.9: Preliminary benchmark results of uniform grid method.

4

Implementation

In this chapter we shall go over the investigation and implementation of the avenues of improvements that we discussed in Section 3.3.2.

4.1. Uniform Grid Method

The uniform grid method is implemented as a new C++ class in the BioDynaMo repository, called the `Grid`. Figure 4.1 shows the UML diagram of the `Grid`, and of the inner class `Box`, which represents a voxel in the uniform grid method. The figure also shows the class diagram of `SoHandle`, which stands for "simulation object handle", and is the unique identifier of a simulation object.

Most noticeable about our implementation is that there is no actual state data of the simulation objects being stored in `Grid` or `Box`. Each `Box` (i.e. voxel) stores only the number of simulation objects it contains (`Box::length_`) and the `SoHandle` of the last simulation object that was added to the voxel. The implementation of `AddObject`, which adds a simulation object to a voxel, updates the `Grid::successors_` vector, which acts like a linked list: the value of a certain element is the index to the next element that is in the `Box`. This avoids the duplication of state data of all the (possible millions) of simulation objects. One could also maintain a vector of simulation object handles per `Box`, but a lot of overhead would go into heap allocations, because the number of objects per `Box` can not be known in advance, but only during runtime.

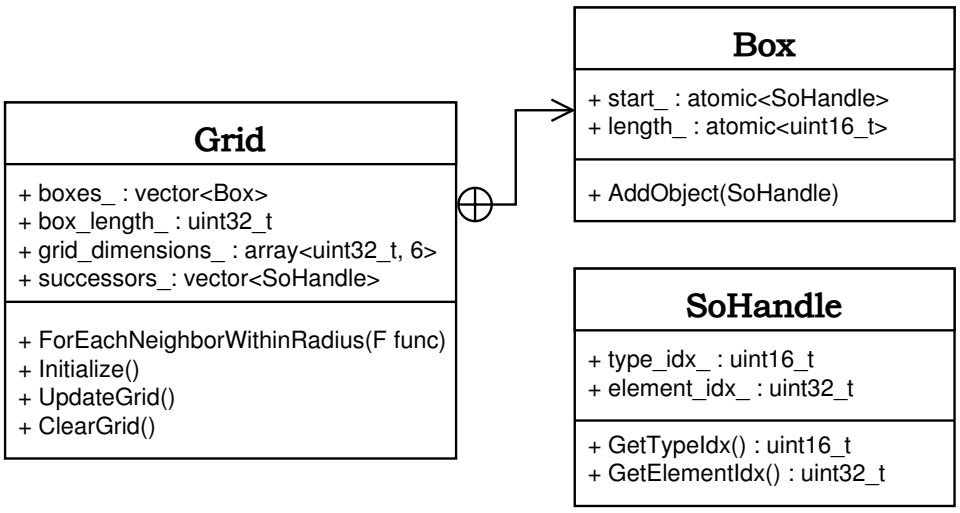


Figure 4.1: UML diagram of the class created for the uniform grid method.

4.1.1. Edge Padding

In the call of `Grid::ForEachNeighborWithinRadius` there is the assumption that each voxel containing simulation objects has 27 surrounding neighbor voxels as explained in Section 3.4.2. At the edges of the simulation space this is of course not the case. For example the voxels at the corners of the space will only have 8 neighboring voxels. In order to prevent the function from illegal memory accesses, we create an extra padding layer of empty voxels around the simulation space. This method should perform better than determining if a voxel is at the edge and adjusting the neighborhood accordingly, as it will introduce unnecessary branching for with every neighborhood operation.

4.1.2. Parallel Build

In the `Grid::UpdateGrid` method, we determine that maximum extent of the simulation space, such that all simulation objects fall within that extent. Then, the edge size of the voxels is determined (based on the size of the largest simulation object), and the corresponding number of voxels are created. Each voxel is populated with simulation objects through the `Box::AddObject` function. This step can however be done in parallel (i.e. using OpenMP), because there is no data dependencies between two consecutive calls. Attention needs to be paid however in updating `Box::length_` and `Grid::successors_`. When multiple threads perform `Box::AddObject` it could be possible that for example both threads try to increment `Box::length_` at the same time. Therefore the data members of `Box` are of type `std::atomic`, and the operations that modify these data members are also atomic.

4.2. GPU Acceleration

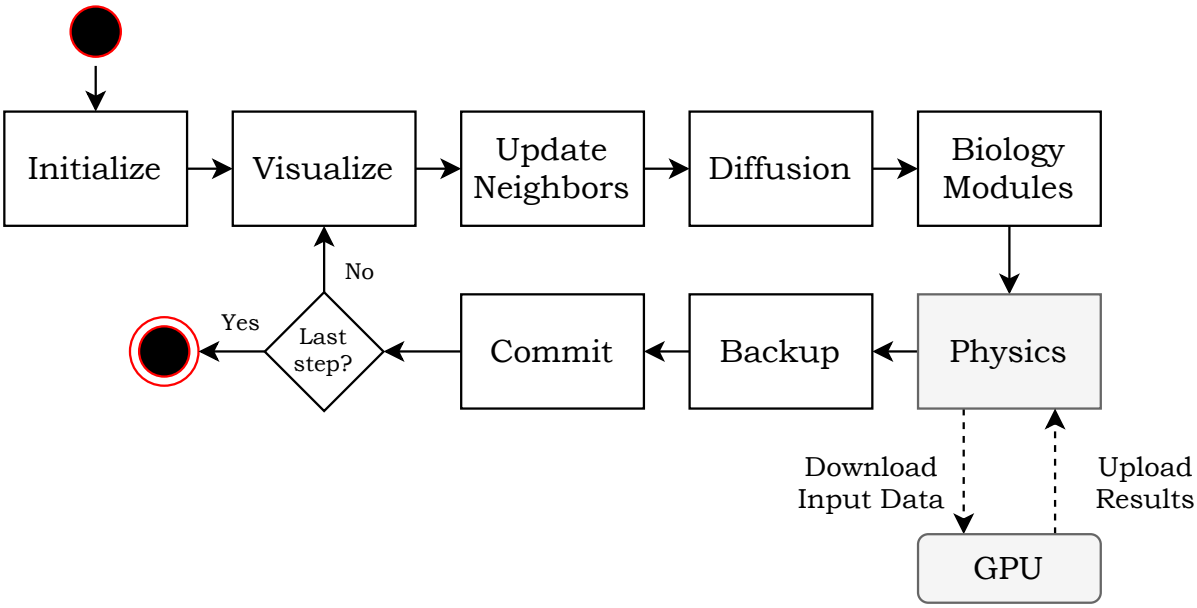


Figure 4.2: Overview of offloading the physics module of BioDynaMo to GPU.

Since the mechanical interactions module is currently the only module in the core of BioDynaMo that makes use of the local neighborhood, we decided to port the uniform grid algorithm as well as the force computation as a single GPU kernel. The main idea is that each GPU thread is handling the physics computations of one cell, which consists of 1) finding the cell's neighborhood, and 2) computing the mechanical forces between the cell and all the cells in its neighborhood. In order to reduce the time to copy data back and forth from the CPU's memory to the GPU's memory, we decided to copy all the required data for a single time step in one time, rather than on a per-cell basis. It is not possible to compute the mechanical interactions for more time steps per data copy, due to the fact that other operations (that happen on the CPU side) will most likely affect the cells' states, and thus introduce time dependencies between data of two consecutive timesteps. Figure 4.2 shows an overview of offloading the physics module on a GPU in BioDynaMo, and Figure 4.3 shows a schematic overview of the physics kernel. We decided to implement the kernel in both OpenCL and CUDA. Some of the

improvements in this section are implemented in code that is isolated from the rest of BioDynaMo, which we shall refer to as *isolated improvements*, while some improvements are merged upstream into BioDynaMo's main repository. The main reason for isolating the physics code from BioDynaMo is to enable a faster development cycle, so that more time could be spent on the research topics that this manuscript addresses. The OpenCL kernel version allows us to try to compile the same code for the future FPGA implementation, while the CUDA kernel version makes it possible to use the more mature tools that Nvidia offers (visual profiling, debugging, etc.). The Initialize GPU step in Figure 4.2 checks if there are any GPUs available on the machine, and in the case of OpenCL additionally compiles the kernel (for CUDA the kernel is precompiled using the nvcc compiler). If no capable GPU is detected, BioDynaMo falls back on the CPU version, and warns the user about this.

4.2.1. Improvement I: Overallocating Buffers

Allocating GPU memory is an expensive operation, and we anticipate that a significant portion of the runtime will be spent on the allocation and deallocation of the memory buffers for the simulation objects each timestep. This anticipation originates from preliminary profiling done on the GPU kernel, where we noticed a significant part of the execution time was consumed by GPU memory allocation. One way to address this problem is to keep the memory buffers alive on the GPU for the duration of a simulation. Obviously this will not be possible if the number of simulation objects increases within a single simulation. For those cases reallocation of memory is necessary. However, in order to not trigger reallocation for every small increment in the number of simulation objects, we allocate slightly larger buffers. For every simulation step we check if the number of simulation objects exceeds the overallocated buffer sizes. If that is not the case, the data can be copied directly from the host to the device. If that is the case, the buffers are deallocated, and we allocate buffers that can hold about X% more simulation objects than the current simulation step. A suitable value for X is to be experimentally found for the simulations BioDynaMo currently can run, and will depend on how large the fluctuations in the number of simulation objects are. An schematic overview of this improvement is given in Figure 4.4. num_{incr} is the percentage increase in number of simulation objects as compared to the previous step.

4.2.2. Improvement II: Reduction in Floating Point Precision

BioDynaMo uses double-precision floating points data types (i.e. doubles) for all its floating point data. However, most consumer GPUs perform stronger in single-precision floating point (i.e. float) operations. This is a manifestation of the fact that the main driving force behind GPU production is the game industry. Game-engines rely mostly on single-precision floating point operations, so GPU manufacturers designed their consumer GPUs with more singe-precision floating point logic units (FP32) than its double-precision counter part (FP64). Some GPU vendors have dedicated cards for high-performance computing, for applications in science for example, that have more FP64 units for double-precision floating point operations. However, since BioDynaMo is targeted to operate (in the near future) on cloud computing infrastructures (with commodity hardware), it will be more beneficial to implement BioDynaMo's GPU code with consumer GPUs in mind. For the sake of accuracy in biological simulations, floats often offer enough precision, which was confirmed by our peers at the Neuroscience Department at Newcastle University. Moreover, a float is half the size of a double 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. This is one of the isolated improvements, as mentioned in the beginning of this section.

4.2.3. Improvement III: Space-Filling Curve Sorting

CUDA and OpenCL organize threads in groups of threads; respectively called blocks and work groups. The execution of the threads on the actual hardware is done in warps (groups of 32 threads), with each warp executing the same instruction, but on different data (i.e. SIMT execution model). The data of the simulation objects is laid down in memory in the order that the 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 a number of scattered memory accesses, which will most likely end up fetching the data from DRAM, which can degrade the performance significantly. This could have been prevented if the simulation objects that are close to each other in space, are also laid down close to each other in memory. This is where

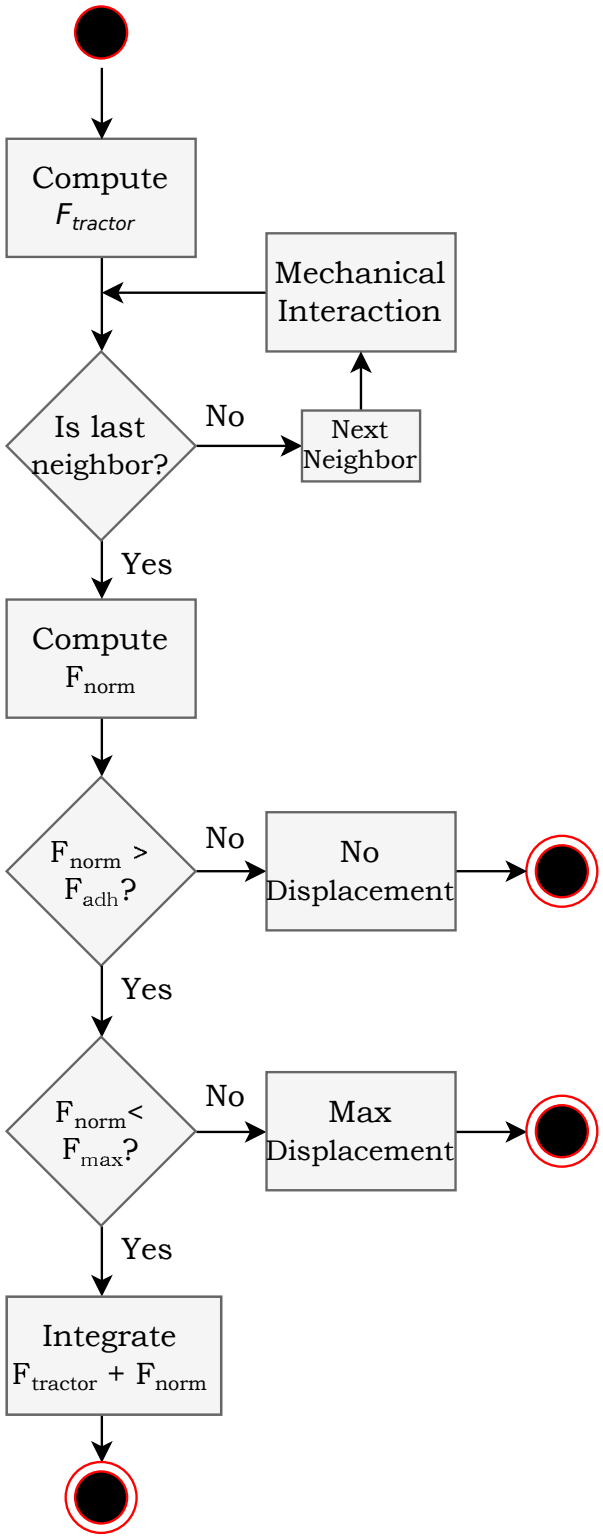


Figure 4.3: Schematic overview of the physics kernel. $F_{tractor}$ is the simulation object's tractor force, F_{adh} its adherence, F_{norm} the normalized total interaction force with the neighbors, and F_{max} the maximum force that can be exerted on a simulation object.

space-filling curves come in; more specifically the Z-order curve. The Z-order curve describes a path in multidimensional space that passes through the data points in a consecutively local order, as illustrated in Figure 4.5. A function that implements a Z-order curve is able to map multidimensional data (such

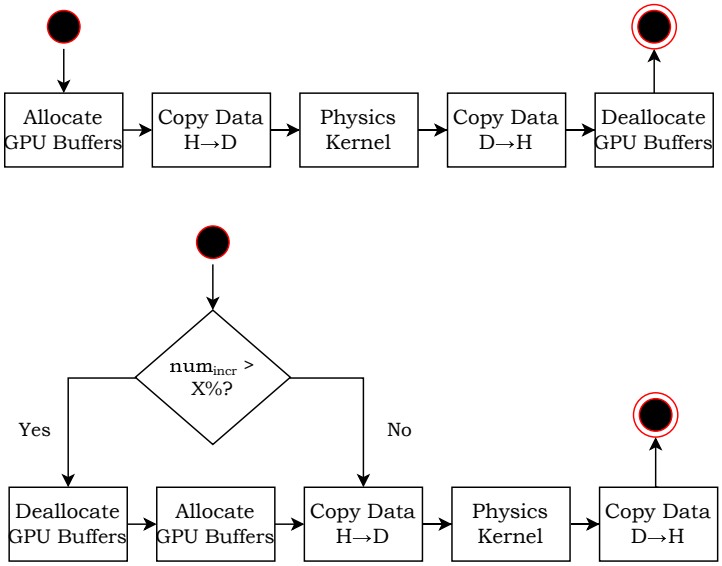


Figure 4.4: Diagram of overallocating buffers on GPU to prevent allocation and deallocation overhead. The top figure shows the original pipeline for comparison. The "Physics Kernel" is shown in Figure 4.3).

as 3D Cartesian coordinates) to a one dimensional array, where consecutive elements of that array are spatially local to each other. 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 simulation objects and sorting their state data accordingly. We expect that the cache line for accessing a simulation object will also contain the data of the simulation objects 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.

Due to the current construction of BioDynaMo, which makes heavily use of template metaprogramming (see Section 2.1.3), we found it more convenient to implement space-filling curve sorting on an isolated version of BioDynaMo. In this isolated benchmark we generate the input data for the mechanical interactions manually to be able to sort the data based on the Z-order curve, and therefore the data sorting was done on the CPU. We are aware that sorting on the CPU might diminish the overall benefits we gain on the GPU, but in future works this could easily be done also on the GPU [19] to avoid the sorting process to become the computation bottleneck.

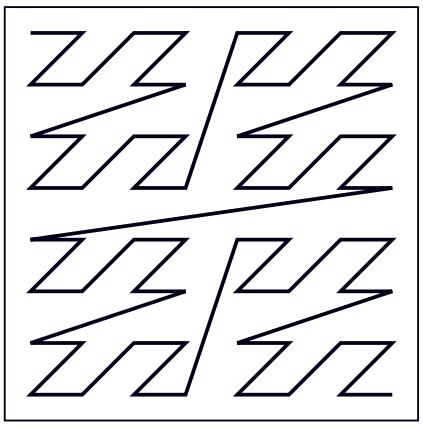
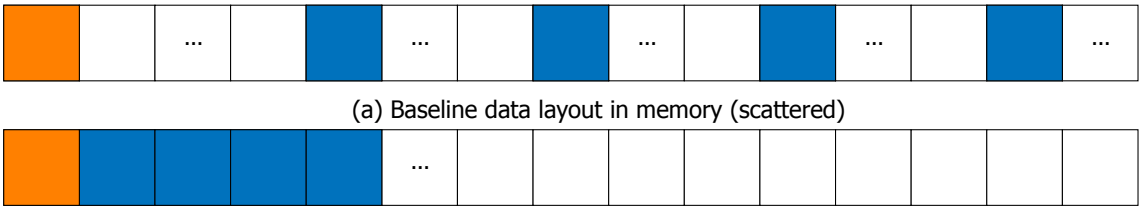


Figure 4.5: The path of a Z-order curve in 2D. Adapted from [3].



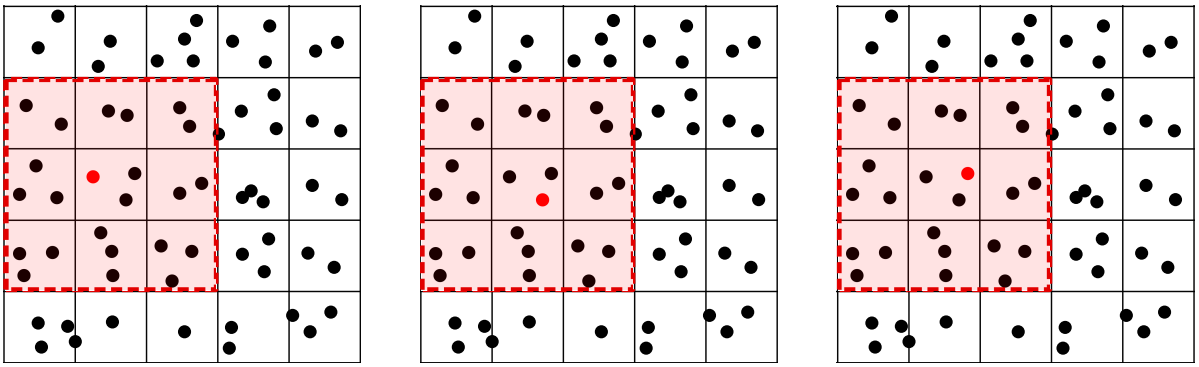
(a) Baseline data layout in memory (scattered)

(b) Sorted data layout in memory (contiguous)

Figure 4.6: The effect on the data layout in memory by sorting on a space-filling curve. The orange square indicates the data of one simulation object, the blue squares the data of its neighbors. The dots indicate an arbitrary amount of data.

4.2.4. Improvement IV: Kernel Redesign to make use of Shared Memory

The concept of each GPU thread handling the physics computations of one cell leaves no 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 work group). The kernel parallelizes the for loop over all cells, so each thread works on data that is independent of the threads in the same block. In order 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 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 exact 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 simulation objects in a single voxel will need to reuse the neighborhood data, which can be stored in shared memory for low-latency memory accesses. The concept is illustrated in Figure 4.7.

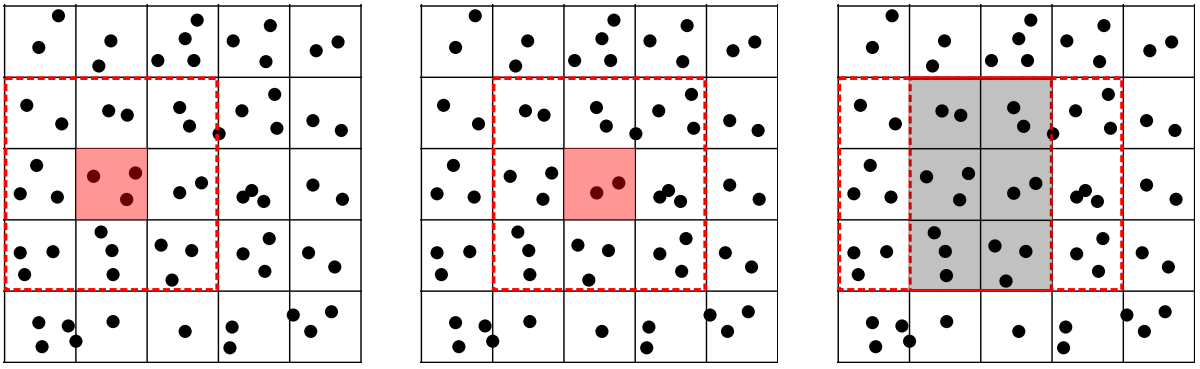


(a) Data required by thread X. (b) Data required by thread Y. (c) Data required by thread Z.

Figure 4.7: Exploiting the reuse of neighboring simulation object data for the usage of shared memory resources on GPU.

4.2.5. Improvement V: Tiling

In Section 4.2.4 we redesigned the kernel execution flow to utilize the shared memory resources on a GPU. Tiling is another technique we can apply to increase the usage of shared memory and increase the throughput even more. Essentially we expand on the idea of placing the neighborhoods that are used by threads that work on simulation objects in the same spatial region in shared memory. Figure 4.8 shows the pattern that lead to the observation of using tiling in BioDynaMo. Figure 4.8a and Figure 4.8b show two threads working on computing the mechanical interaction forces of the cells in the center voxels (the red squares), requiring the data of surrounding voxels (highlighted with a red line). These two threads have a large overlap in the data that is required for the computations as indicated in Figure 4.8c. If we would place the threads that are responsible for the voxels that are near to each other in the same CUDA block (or OpenCL work group), it would allow us to put more relevant data in shared memory and reduce the latency of more memory accesses. Figure 4.8 shows the 2D projection of the actual 3D simulation space, so the tiles would in reality be cubes, instead of squares, which would lead to even more overlap in shared data within a tile.



(a) Data required by thread X. (b) Data required by thread Y. (c) Data overlap between threads X and Y.

Figure 4.8: Exploiting data overlap between multiple GPU threads for the use of shared memory.

The challenge with tiling is to find a suitable size of the tiles. Putting too little voxels in a tile will not improve the performance, but there is also a limit to the shared memory capacity, so putting too many voxels in a tile will also not be beneficial. Due to time limitations this technique has not been implemented, but we mention the concept for possible future works.

4.3. FPGA Acceleration

As discussed in Section 2.2.2 there are many ways of programming an FPGA for general purpose computing. We choose to use the OpenCL framework for the following reasons: 1) it abstracts the lower level hardware description languages to allow for rapid development, 2) it allows for a design space exploration that comes with the HLS toolkit that OpenCL relies on, and 3) we already use OpenCL to program the GPU kernel mentioned in Section 4.2, so it fits the already existing heterogeneous execution model. Since OpenCL is portable among different architectures, we use the initial GPU kernel to be the baseline kernel for the FPGA. Currently, only Intel (formerly Altera) and Xilinx provide an OpenCL implementation for FPGAs. We chose to base the FPGA implementation on the Intel toolkit mainly because of the absence of support for other toolkits in the computing resources available to us.

Due to the way parallelism is handled in FPGAs in contrast to GPUs it is to be expected that the performance of the baseline kernel will be very low. As mentioned in Section 2.2.2 FPGAs inherently makes use of pipeline parallelism, rather than the SIMD-style of parallelism GPUs make use of. An OpenCL kernel that is written for GPUs might be portable to FPGAs in terms of compilation, but the performance is most likely not portable. In order to utilize the architectural features that FPGAs offer, we need to redesign the OpenCL kernel accordingly. The steps involved in the design flow of an OpenCL kernel for FPGAs is given in Figure 4.9.

4.3.1. Single work-item kernel vs. NDRange kernel

The first design choice that needs to made is whether to design a single work-item kernel or a NDRange kernel. Based on the information provided in [20] a NDRange kernel is preferred if there are no memory or loop dependencies within the kernel, because it allows the aoc compiler to automatically schedule the work-items (i.e. threads) efficiently in parallel. One disadvantage of choosing a NDRange kernel over a single work-item kernel is the lack of information one can get from the Intel HLS Report after compiling the kernel (but before synthesis). The reason is that the Intel Offline Compile can infer only from the kernel code what the impact of each optimization will be on the synthesized design. An NDRange kernel expresses pipeline parallelism implicitly from the thread organization that one launches the kernel with, while in a single-task kernel the developer needs to explicitly program the kernel in such a way that pipeline parallelism is achieved. Due to the fact there are no memory and loop dependencies in the mechanical interaction kernel, we choose to implement the FPGA kernel as a NDRange kernel. We shall refer to this state of the kernel as the *baseline* implementation. Section 4.3.1 shows the resource utilization of the kernel after synthesis, as obtained with the Quartus II Prime HLS software. With the Quartus software we were also able to obtain the physical layout of the FPGA, and how the

Fitter Summary (baseline)	
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Pro Edition
Revision Name	top
Top-level Entity Name	top
Family	Arria 10
Device	10AX115N3F40E2SG
Timing Models	Final
Logic utilization (in ALMs)	76,328 / 427,200 (18 %)
Total registers	135409
Total pins	335 / 826 (41 %)
Total virtual pins	0
Total block memory bits	7,068,872 / 55,562,240 (13 %)
Total RAM Blocks	610 / 2,713 (22 %)
Total DSP Blocks	74 / 1,518 (5 %)
Total HSSI RX channels	16 / 48 (33 %)
Total HSSI TX channels	16 / 48 (33 %)
Total PLLs	60 / 112 (54 %)

Table 4.1: Summary of the fitting phase of the FPGA baseline implementation.

resources are laid out on the chip, as shown in Figure 4.10. As can be seen from the table and the figure, most of resources are not utilized, because no FPGA-specific optimizations have been applied. We expect the baseline implementation to perform poorly due to this fact. The dark-purple-colored region in Figure 4.10 represents the Board Support Package (BSP) that is required for OpenCL kernels to be executed. The BSP takes a fixed percentage of the FPGA's resources, which are included in the numbers in Section 4.3.1.

4.3.2. Improvement I: Convert Nested Loops to Single Loop

In the mechanical interactions kernel there is a triple-nested for loop that goes over the candidate neighbor simulation objects in the surrounding 3x3x3 voxels. According to [20] nested for loops generally create a larger memory footprint than single loops. In our case a single for loop can be realized through the use of a look-up table (LUT). Let's take a closer look at the nested for loop:

```
for (int z = -1; z <= 1; z++) {
  for (int y = -1; y <= 1; y++) {
    for (int x = -1; x <= 1; x++) {
      uint32_t bidx = CalculateBoxId(my_box_coords + make_int3(x, y, z));
      if (!EmptyBox(bidx)) {
        ComputeForces(...);
      }
    }
  }
}
```

Listing 2: Nested for loop in OpenCL kernel.

Each thread is faced with the nested loops as shown in Listing 2. In the inner-most loop the voxel id of the neighboring boxes is obtained from the voxel coordinates with respect to the voxels coordinates of the simulation object the current thread is working on. Since the offset to the neighboring voxels is always the same, it is possible to map them to a look-up table, such as the one shown in Listing 3. Although this seems like a minor adjustment, the implications could be significant, since it affects all simulation objects.

```
const int3 offset[27] = {
    make_int3(-1, -1, -1), make_int3(0, -1, -1), make_int3(1, -1, -1),
    make_int3(-1, 0, -1),  make_int3(0, 0, -1),  make_int3(1, 0, -1),
    make_int3(-1, 1, -1),  make_int3(0, 1, -1),  make_int3(1, 1, -1),
    make_int3(-1, -1, 0),  make_int3(0, -1, 0),  make_int3(1, -1, 0),
    make_int3(-1, 0, 0),   make_int3(0, 0, 0),   make_int3(1, 0, 0),
    make_int3(-1, 1, 0),   make_int3(0, 1, 0),   make_int3(1, 1, 0),
    make_int3(-1, -1, 1),  make_int3(0, -1, 1),  make_int3(1, -1, 1),
    make_int3(-1, 0, 1),   make_int3(0, 0, 1),   make_int3(1, 0, 1),
    make_int3(-1, 1, 1),   make_int3(0, 1, 1),   make_int3(1, 1, 1)};

for (int i = 0; i < 27; i++) {
  uint32_t bidx = CalculateBoxId(my_box_coords + offset[i]);
  if (!EmptyBox(bidx)) {
    ComputeForces(...);
  }
}
```

Listing 3: Single for loop with look-up table in OpenCL kernel.

4.3.3. Improvement II: Loop Unrolling

One way to improve the performance of an OpenCL kernel for FPGAs is loop unrolling. Unrolling a loop inside a kernel reduces the number of iterations required to complete the loop. When a loop is selected for loop unrolling the OpenCL Offline Compiler duplicates the hardware that is required for the computations. This relaxes the resource constraints that are put on the high-level synthesis compiler, and enables more pipeline parallelism. In the Intel OpenCL SDK for FPGAs this is done by placing a `#pragma unroll` statement above the for loop. As illustrated in Figure 4.3, consecutive iterations of this loop do not depend on each other. Therefore, the force between two simulation objects is independently calculated from the force of two other simulation objects. The absence of data dependencies leads us to exploit pipeline parallelism, because consecutive iterations of the loop can be scheduled with a low initiation interval (Section 2.2.2).

There is however a limit to how much a loop can be unrolled, due to the fact that the FPGA fabric has a limited number of resources. To warn developers how the expected usage utilization, the Intel OpenCL for FPGA SDK includes an tool that performs a pessimistic resource estimation. The Intel OpenCL for FGPA SDK (similar to the Xilinx OpenCL SDK) allows for partial loop unrolling. Section 4.3.3 shows the resource utilization after the fitting stage for this version of the mechanical interactions, with the loop in question unrolled 32 times. From Figure 4.11 we can clearly see the change in chip utilization.

4.3.4. Improvement III: Coalesced Global Memory Accessing

From the Intel HLD reports we could observe that the memory accesses are not coalesced. One of the unique advantages of FPGAs is that hardware that is responsible for reading and writing to the main memory is reconfigurable, and can be adjusted to the requirements of the application. Coalesced memory access can be identified in the Intel HLD report by hovering over the load (LD) and store (ST) units. Figure 4.12 shows 8× LD units accessing data with a single float per unit at a time. The figure also shows a LD unit accessing eight floats at a time as indicated by the red circle. In order for the Intel Offline Compile to generate wider LD and ST units, the application is required to access the data in a strided pattern. In the mechanical interactions kernel the data that needs to be accessed is the simulation objects' state data. However, since we perform the operations until now in a similar fashion as we did in the GPU kernel, the access pattern is not strided. Remember that for each simulation object that is being processed, the neighbor cells need to be accessed, whose indices are scattered. As shown in Listing 2 the indices are calculated by a regular offset from the voxel indices of the simulation object in question. Even though we would apply the Z-order curve sorting on the data as we did in Section 4.2.3, the Intel Offline Compile could not infer from the OpenCL kernel code that the accesses should coalesced (since the data layout is only known at runtime).

Fitter Summary (improvement II)	
Quartus Prime Version	16.1.2 Build 203 01/18/2017 SJ Pro Edition
Revision Name	top
Top-level Entity Name	top
Family	Arria 10
Device	10AX115N3F40E2SG
Timing Models	Final
Logic utilization (in ALMs)	193,963 / 427,200 (45 %)
Total registers	394584
Total pins	335 / 826 (41 %)
Total virtual pins	0
Total block memory bits	16,257,544 / 55,562,240 (29 %)
Total RAM Blocks	1,925 / 2,713 (71 %)
Total DSP Blocks	1,134 / 1,518 (75 %)
Total HSSI RX channels	16 / 48 (33 %)
Total HSSI TX channels	16 / 48 (33 %)
Total PLLs	60 / 112 (54 %)

Table 4.2: Summary of the fitting phase of the FPGA improvement II.

The solution that we propose to circumvent this issue, is to reorganize the data structures that contain the simulation objects' state. Instead of passing to the OpenCL kernel the raw state data, we prepare the data on the CPU in such a way that the data can be accessed in a strided fashion, and Intel Offline Compiler can generate the required hardware for coalesced memory accesses. Essentially, the data preparation moves the operations of the uniform grid method to the CPU. Recall that the mechanical interactions calculate the forces between two simulation objects. By creating a data structure where the state data of two simulation objects that are supposed to collide are contiguous to each other in memory, the indexing of the data structures can be strided. The strided access pattern should be picked up by the Intel Offline Compiler for the generation of custom hardware that can access the memory objects in a coalesced manner. Effectively, this should reduce the number of accesses to global memory, and thus increase the throughput of data, and feed the computational pipeline with the required data.

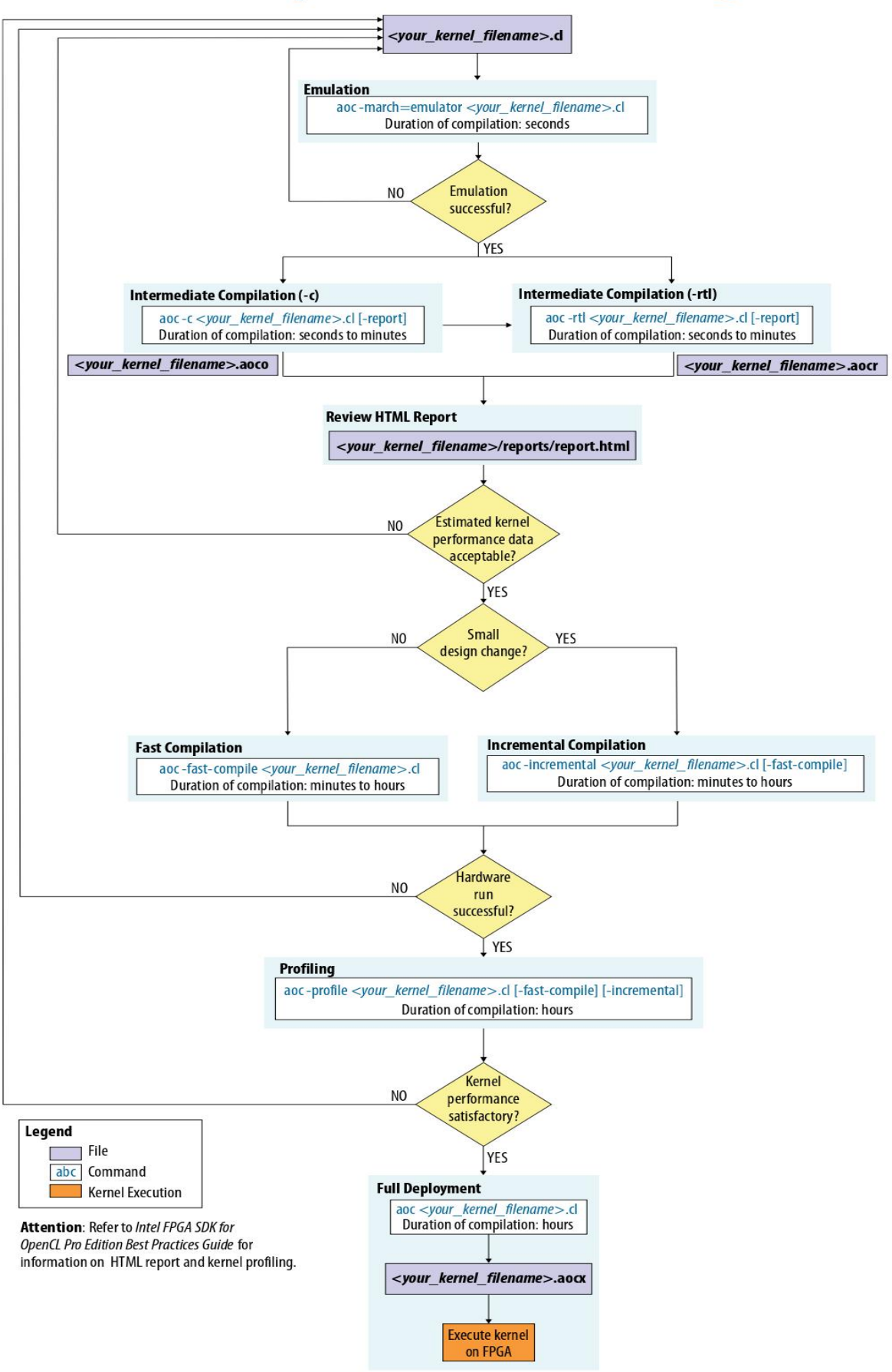


Figure 4.9: Multistep Intel® FPGA SDK for OpenCL™ Pro Edition Design Flow [].

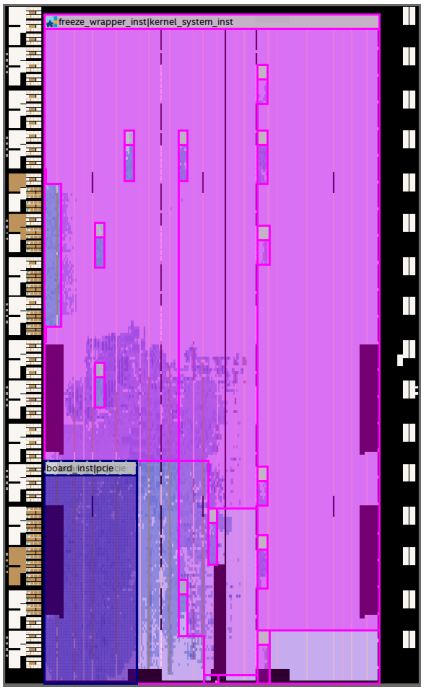


Figure 4.10: Overview of the synthesized baseline implementation on the FPGA chip, as generated by the Quartus Chip-Planner tool.

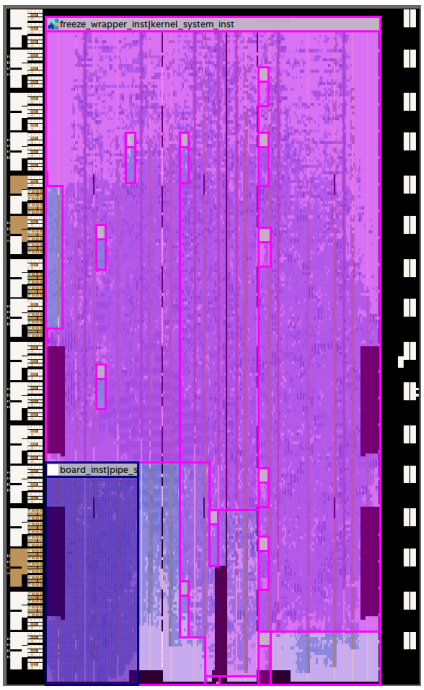


Figure 4.11: Overview of the synthesized FPGA Improvement II implementation on the FPGA chip, as generated by the Quartus Chip-Planner tool.

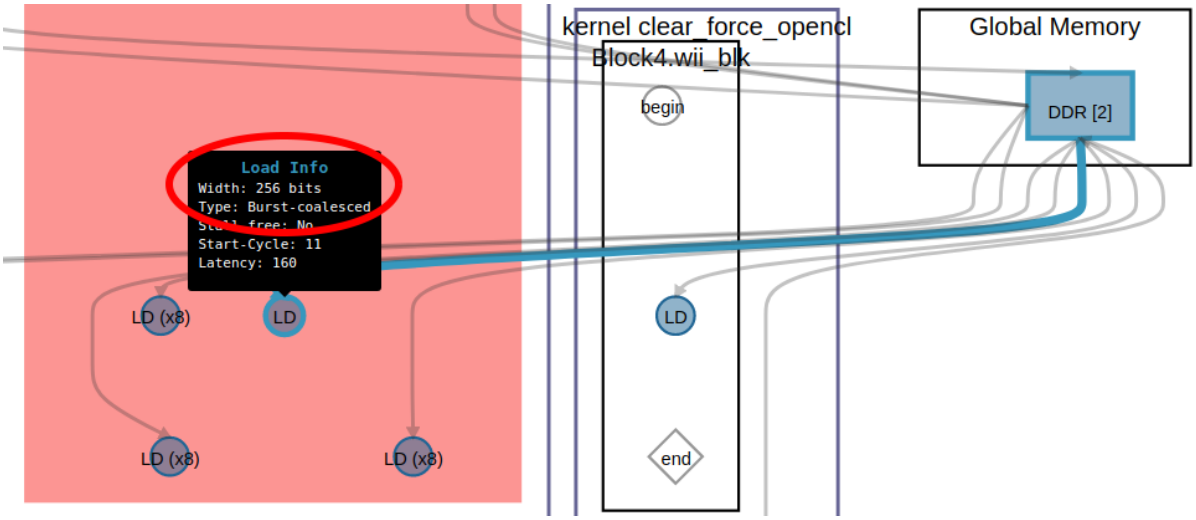


Figure 4.12: FPGA load unit for coalesced memory access as generated by the Intel HLD FPGA Report.

5

Evaluation

In Chapter 4 we described the implementation of the performance improvements proposed in Section 3.3.2. This chapter will cover the evaluation of each implementation, and how each improvement affects the performance of BioDynaMo.

5.1. Experimental Setup

The hardware on which the evaluations are done belong to the CERN IT department, and are tabulated in Table 5.1 and Table 5.2.

The CPU of both systems consist of two physical sockets, with each 20 cores, in a non-uniform memory access (NUMA) design. In order to have a good understanding of scaling up with the number of threads on one machine, we decided to run the benchmarks on only one of the sockets, and effectively removing the NUMA effects (e.g. false sharing) that would otherwise play a role in the timing results. In practice this was achieved by using the Linux utility tool `taskset`. This decision also implies that only 20 out of the 40 cores were utilized for the performance benchmarks. For profiling benchmarks that were run on the CPU, we made use of Intel VTune, which uses hardware counters to sample the call stack periodically.

For the GPU benchmarks we use the Nvidia SDK visual profiler nsight, which internally relies on nvprof. The CUDA profiling tools are far more mature and user-friendly than the OpenCL profiling tools. Prior to capturing the timing data for profiling CUDA or OpenCL kernels, we run a number of 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.

As mentioned before we use the Intel OpenCL SDK for FPGAs for the FPGA setup. Internally it relies on the Quartus HLS toolkit for the synthesis, fitting and routing of the FPGA from the OpenCL kernel. In [20] it is mentioned that the Intel OpenCL SDK for FPGAs has a possibility of integrating hardware counters in the board design, at the cost of a small performance penalty. In our designs we timed the total runtime with and without profiling enabled, but found no significant performance difference, and therefore chose to enable profiling by default (which is set by the aoc compiler flag `--profile`).

Table 5.3 shows the description of parameters we use in the evaluation. The benchmark used for evaluating the performance of each of the version of the mechanical interactions module is described as follows. A cubic grid ($dim_x = dim_y = dim_z$) of somata cells with initially all equal sizes, are spawned in an overlapping manner, such that the first 100 timesteps are completely governed by mechanical interactions. We will time the first 100 timesteps ($T_s = 100$) and take the average to be the execution time per timestep that will be shown in the upcoming results. Throughout this chapter references to footnotes lead to the Github link to the specific branch containing the benchmark that was executed, allowing the reader to reproduce the benchmarks.

Specification	Nvidia GTX1080 Ti System
GPU Chip	4x Nvidia GTX1080 Ti
GPU RAM	11GB GDDR5X
GPU RAM Bandwidth	484 GB/s
Max. Power Consumption	250W (GPU)
CPU	Intel(R) Xeon(R) CPU E5-2640
CPU cores	40 (2 sockets)
CPU DRAM	256GB DDR4

Table 5.1: Specifications of the system with the GPU accelerator.

Specification	Nallatech 385A FPGA System
FPGA Chip	Intel Arria 10 GX 1150 GX
FPGA UGRAM	2GB DDR4 + 2GB DDR3
Max. Power Consumption	75W (FPGA)
CPU	Intel(R) Xeon(R) CPU E5-2630
CPU cores	40 (2 sockets)
CPU DRAM	64GB DDR4

Table 5.2: Specifications of the system with the FPGA accelerator.

Parameter	Description	Unit
$num_{objects}$	The number of simulation objects	N/A
$num_{threads}$	The number of CPU threads	N/A
T	Number of timesteps	N/A
$dim_{x\|y\|z}$	The dimensions of the simulation space	Model-specific
buf_{alloc}	Percentage of overallocating GPU buffers	%

Table 5.3: Parameters description

5.2. Performance Overview

Figure 5.1 shows an overview of the execution times of each version of the mechanical interaction module in BioDynaMo. Note that the x-axis is on a logarithmic scale. The order follows from the order in which the versions were introduced in Chapter 4. The consecutive GPU or FPGA versions include the implementation of the prior version, so for example GPU Improvement II includes the changes made for GPU Improvement I, and the same for the FPGA versions. Figure 5.2 shows the speedup of each version with respect to the baseline version.

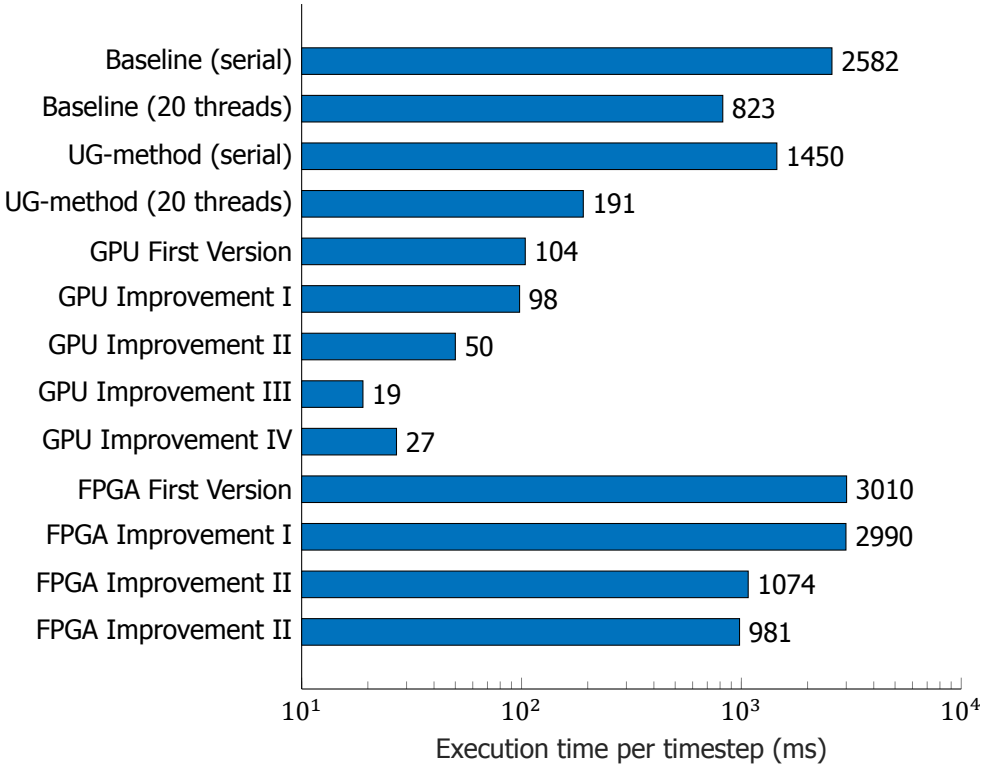


Figure 5.1: The execution time per simulation step for all mechanical interaction implementations, with $num_{objects} = 2097152$.

5.3. Uniform Grid Performance

The first design choice we made was to replace the kd-tree radial neighbor search method with the uniform grid (UG) method as explained in Section 4.1. In Figure 3.9 shows the results of a preliminary benchmark on an the initial implementation of the UG method [1].

5.3.1. Uniform Grid Method: Scaling with Number of Simulation Objects

We further evaluate the performance of the uniform grid method by measuring the execution time for the radial neighborhood search as the number of simulation objects in a simulation increases. For reference we perform the same measurements for the baseline implementation, which is the kd-tree method. Figure 5.3 shows results of these measurements. We observe that the UG-method scales linearly with the number of simulation objects, in accordance to our analysis in Section 3.4.2, and the execution time grows with a smaller gradient than the baseline method (kd-tree).

5.3.2. Uniform Grid Method: Scaling with Number of Threads

Figure 5.4 shows how the UG method scales with the number of threads on a single machine. We observe that the UG-method scales stronger with the number of threads, compared to the kd-tree method. Amdahl's law (Equation (2.1)) states that the speedup gained from multithreaded execution

[1] https://github.com/BioDynaMo/biodynamo/tree/thesis-bench-sdmethod

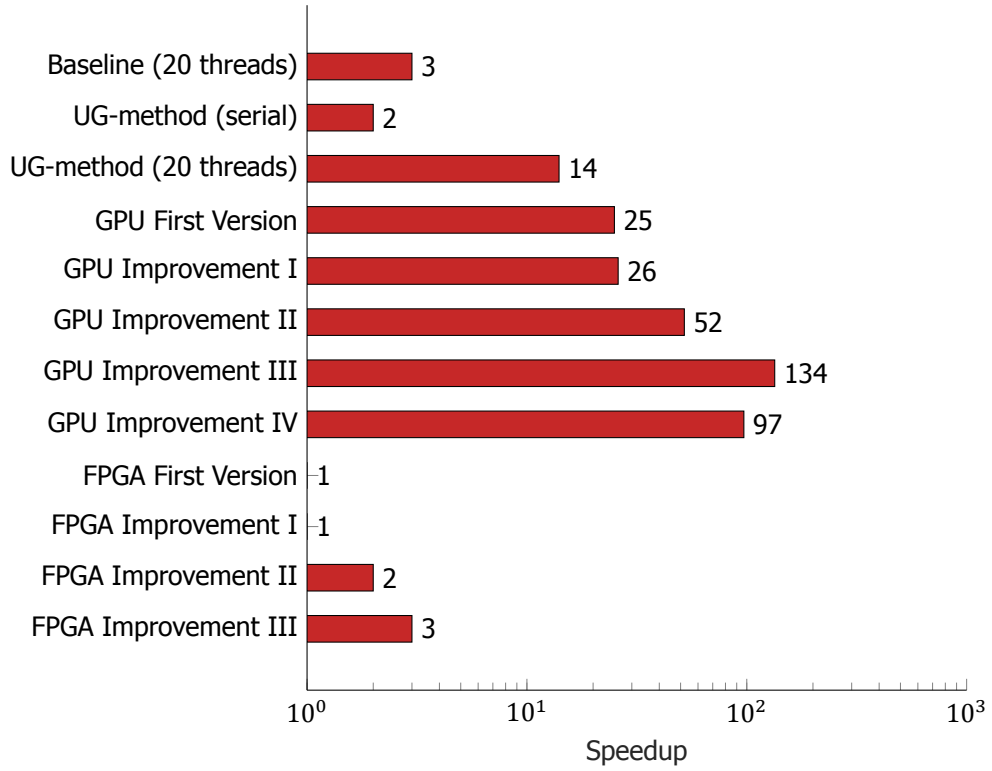


Figure 5.2: The speedup per simulation step with respect to the serial baseline version for all mechanical interaction implementations, with $num_{objects}$ = 2097152.

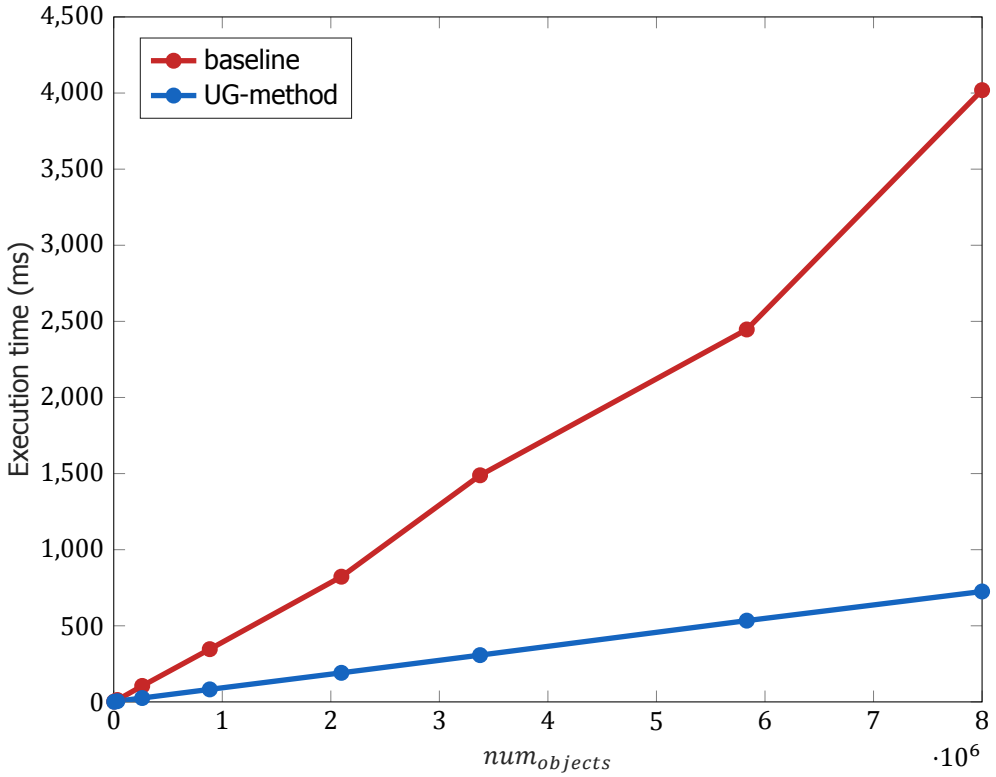


Figure 5.3: Execution time of radial neighborhood search with varying number of simulation objects.

is determined by the parallelizable part of the code. Building the kd-tree structure is a serial operations, while on the other hand, with the UG method it is possible to parallelize the building phase. This implies that a larger fraction of the code of UG method is parallelizable, and thus enables a higher attainable speedup than the kd-tree method.

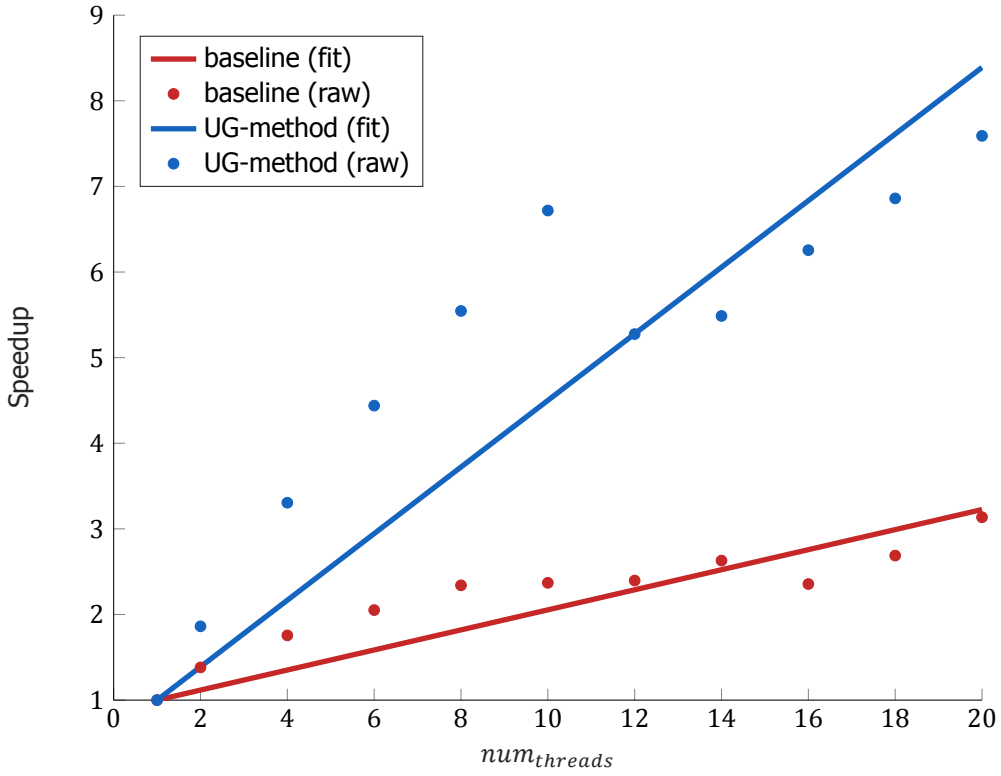


Figure 5.4: Execution time of radial neighborhood search with varying number of threads.

5.4. GPU Performance

In this section we will discuss the results and findings from the evaluation of the proposed GPU improvements from Chapter 4 in more detail. Some of the upcoming subsections will contain benchmarks more tailored to highlight the benefits of the improvement in more specific use cases or simulations. We will follow the same order as in Chapter 4.

The first version of the GPU implementation [2] already offers an order of magnitude in speedup as compared to the serial baseline version, and is significantly faster than its multithreaded CPU variant, the UG-method on 20 threads (64% faster). This is a remarkable results, as the price for the CPU on the Nvidia GTX1080 Ti System is about three times the price of the GPU card itself. Any further speedups gained from the improvement will further justify the use of GPUs for the BioDynaMo runtime.

5.4.1. Discussion on GPU Improvement I

Overallocating GPU buffers appears to have less of an impact on the performance than we originally anticipated. Upon further investigation it seems that our preliminary profiling efforts (as mentioned in [**?**] were skewed. The preliminary benchmarks were targeting executions of single timesteps, rather than of multiple timesteps. In Figure 5.5 shows the profile that initially made us believe that memory allocations are relatively expensive.

Figure 5.6 on the other hands shows a part of the profile of longer-running simulations with multiple timesteps [3]. Except for the first cudaMalloc call, the consecutive calls appear to be much less time-consuming. From [15] we find that the first runtime function, that is not a device management or

[2]https://github.com/Senui/biodynamo_physics/tree/thesis-bench-gpu-initial-and-sorted
[3]https://github.com/BioDynaMo/biodynamo/tree/thesis-bench-overalloc

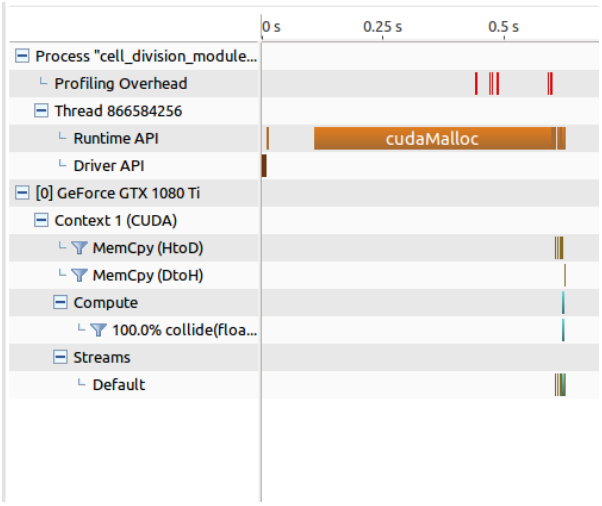


Figure 5.5: Incorrect assumption made based on single timestep profiles.

version function, will lead to an implicit initialization of the CUDA runtime, and that this could manifest itself in timing data. From this fact we can deduce the the cudaMalloc call in Figure 5.5 is the first CUDA runtime function, and therefore the timing data includes the runtime initialization time. Consequently, this explains the observation that the performance gained from improvement I is not as high as initially anticipated.

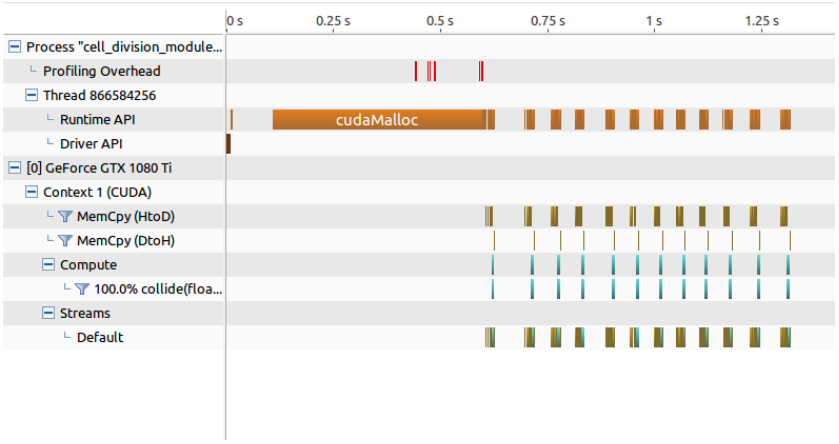


Figure 5.6: Profile of simulating a longer-running simulation, and the observation on GPU memory allocation time.

To quantify the performance gained by buffer overallocation, we profiled long-running simulations in which the simulation objects rapidly grow and divide, and therefore require increasingly larger GPU buffers as the simulation progresses. This type of simulation was chosen to emphasize the potential benefits that can be obtained from overallocating buffers, and represents biological processes that are very dynamic and for example exhibit strong developmental tissue growth. We used nvprof to get the numerical timing results showed in Section 5.4.1. From the results we can see that buffer overallocation saves about 4.5% with respect to the entire GPU runtime.

5.4.2. Discussion on GPU Improvement II

From Figure 5.1 we can see about a 2×speedup gained from reducing the data types that define a cell's state from doubles to floats [4][5]. For the Nvidia GTX1080 Ti GPU the performance ratio of double-precision floating point operations to the single-precision floating point operations can be described

[4]https://github.com/BioDynaMo/biodynamo/tree/thesis-bench-doubles2
[5]https://github.com/BioDynaMo/biodynamo/tree/thesis-bench-floats

Type	Time(%)	Time	Calls	Avg	Name
GPU activities:	91.79%	109.903s	100	1.09903s	my_kernel
	6.29%	7.53401s	1600	4.7088ms	[CUDA memcpy HtoD]
	1.92%	2.29412s	100	22.941ms	[CUDA memcpy DtoH]
API calls:	86.84%	109.904s	100	1.09904s	cudaDeviceSynchronize
	7.90%	10.0035s	1700	5.8844ms	cudaMemcpy
	4.43%	5.60520s	878	6.3841ms	cudaFree
	0.81%	1.02469s	895	1.1449ms	cudaMalloc

Table 5.4: Nvprof profiling results without buffer overallocation.

Type	Time(%)	Time	Calls	Max	Name
GPU activities:	91.97%	110.420s	100	2.15402s	my_kernel
	5.95%	7.13945s	1600	40.227ms	[CUDA memcpy HtoD]
	2.08%	2.50238s	100	45.239ms	[CUDA memcpy DtoH]
API calls:	91.86%	115.245s	100	4.30787s	cudaDeviceSynchronize
	7.81%	9.80477s	1700	45.633ms	cudaMemcpy
	0.28%	354.78ms	35	332.02ms	cudaMalloc
	0.03%	36.802ms	18	4.6759ms	cudaFree

Table 5.5: Nvprof profiling results with buffer overallocation ($buf_{alloc} = 100\%$).

as $\frac{FP64}{FP32} = \frac{1}{32}$ [21]. This ratio implies that about 32×more single-precision floating point operations can be performed than double-precision floating point operations. Note that the speedup we expect is not 32×, due to the fact that there are other instructions being executed by the GPU, and the relative latency among these instructions varies heavily.

5.4.3. Discussion on GPU Improvement III

Sorting the simulation objects' state date based on a space-filling curve proved to reduce the execution time significantly, namely by a factor of 134× in comparison to the serial baseline implementation [6]. This speedup confirms that the GPU kernel enjoys more spatial and temporal cache locality when sorted, than when not sorted. Note that this speedup does not include the sorting time which is done on the CPU (as mentioned in Section 4.2.3. We are however convinced from the findings of [19] that once the sorting is implemented on the GPU, the overall speedup will not differ significantly from the current value.

5.4.4. Discussion on GPU Improvement IV

Recall that this improvement [7] introduced a redesign of the GPU kernel in order to utilize the shared memory resources. This improvement however appears to perform worse than the previous kernel version, which does not make use of shared memory. One of the reason we believe the kernel performs worse is because of the introduction of atomic operations in the kernel. The use of atomics is necessary in order to build up the shared data structures in parallel, however this will lead to threads locking each other out from the shared data structures that they attempt to modify. Moreover, the kernel needs to perform boundary checks on the blocks (CUDA) or work groups (OpenCL) that are being executed by the GPU. Recall that in this version of the kernel, we execute the operations per uniform grid voxel, and need to access the data of the 27 surrounding voxels. Accessing the 27 surrounding voxels is done through the block ID (or work group ID)

[6] https://github.com/Senui/biodynamo_physics/tree/thesis-bench-gpu-initial-and-sorted
[7] https://github.com/BioDynaMo/biodynamo/tree/improve-gpu-code

5.5. FPGA Performance

In this section we will discuss the results and findings from the evaluation of the proposed GPU improvements from Chapter 4 in more detail. Some of the upcoming subsections will contain benchmarks more tailored to highlight the benefits of the improvement in more specific use cases or simulations. We will follow the same order as in Chapter 4.

The first version of the OpenCL kernel for FPGA [8] perform worse than the baseline serial version. The main reason being that we do not benefit from any of the advantages the FPGAs offer (over CPU / GPU). The evaluation mainly proves the fact that the performance of OpenCL kernels is not portable among different hardware accelerators.

5.5.1. Discussion on FPGA Improvement I

The use of a lookup-table [9] showed only little performance gain, in comparison to the first version of the kernel. This is mainly due to fact that there has not been duplication of resources yet, and therefore we only gain from the benefits of using a LUT *once*. The performance benefits of the LUT improvement will propagate onward to the next improvement. The reduction in resource utilization, as displayed in Section 4.3.3, was according to expectations, and will also be prove to be more beneficial once loop unrolling has been performed.

5.5.2. Discussion on FPGA Improvement II

Loop unrolling has a significant impact on the performance of the FPGA implementation [10], and improved the performance by 64%, performing better than the serial UG-method, and almost on par with the multithreaded baseline version . The Intel OpenCL for FPGA SDK has the option to profile the OpenCL kernel. Profiling can be selected by adding the `--profiling` option to the aoc compiler. Upon executing the kernel, a profile log file is generated, and can be opened for visual display. In the profile log of this version of the kernel, the visual profiler showed that the bandwidth from the global memory to the on-chip resources is the performance bottleneck, and is due to the fact that the access pattern is not strided, and therefore the memory access are not coalesced into one large memory access.

5.5.3. Discussion on FPGA Improvement III

In this version of the OpenCL kernel for the FPGA we tried to address the absence of coalesced memory accesses, by redesigning the data structures [11], as explained in Section 4.3.4. As shows in Figure 5.1 the overall speedup is not impressive. The actual time spent in the kernel execution (including data transfer time) is in fact on average 170 ms, which would have lead to a speedup of 15× in comparison to the serial baseline version. However, the time it takes to prepare the new data structures (on the CPU), is significantly higher than the kernel execution time, and therefore cancels out the potential benefits that one could gain from coalesced memory accesses.

[8] https://github.com/Senui/bdm-fpga/commit/bc8387f927f7ef81a011fd05b662e5609eba211d
[9] https://github.com/Senui/bdm-fpga/commit/baf38061b282732ebfd8928e1d8c9945972f3a83
[10] https://github.com/Senui/bdm-fpga/commit/4f95cd4731bbabeb93aa6e455c79e553a255e4f8
[11] https://github.com/Senui/bdm-fpga/commits/batched-kernel

6

Conclusions

This chapter concludes the work and the research that was conducted in this manuscript. In Section 6.1 the contributions of this work will be highlighted and briefly discussed. Then, in Section 6.2 we present our thoughts on the future work that could emerge from these efforts, and are related to similar research.

6.1. Contributions

The goal of this thesis was to perform a comparative study of the acceleration potential of GPUs and FPGAs of BioDynaMo's core to enable fast simulation of large-scale and complex biological models. At the core, BioDynaMo has several functional modules that are ready to be used by the life science community. In order to find out 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 module was the computational bottleneck by a large margin, and therefore we decided to focus our efforts in accelerating BioDynaMo to that specific module. To this extent we investigated three avenues of improvements: 1) algorithmic alternatives, 2) GPU acceleration, and 3) FPGA acceleration. Within each of these avenues we made efforts in optimizing the performance that can be gained: 1 additional algorithmic improvement, 4 additional GPU improvements, and 3 additional FPGA improvements, leading to a total of 8 different versions of the mechanical interactions module.

The main reason why the mechanical interactions module was the computation bottleneck of BioDynaMo lies in the fact that it requires the knowledge of locality (i.e. the neighborhood of simulation objects) to be present. We found an alternative to the kd-tree method, the uniform grid (UG) method that was implemented in the baseline version of BioDynaMo that proved to be an excellent candidate for exploiting performance improvements. Not only did the UG method perform faster on systems with only CPUs on board, but it also opened up possibilities to exploit the advantages of systems with hardware accelerators such as GPUs and FPGAs. More concisely the following contributions were made:

- We implemented the uniform grid (UG) method as an alternative to the kd-tree method for radial neighborhood search that are used within the mechanical interactions module. Due to the nature of the UG method it was possible to parallelize a larger portion of the code, in comparison to the kd-tree method. A 2×speedup was achieved when comparing the serial versions of both methods, while a 5×speedup was achieved when comparing the multithreaded versions (20 threads on 20 physical cores on 1 CPU socket).
- A GPU kernel is developed that allows us to offload the mechanical interactions computations to a GPU. The initial version of the kernel lead to a 25×speedup to the serial baseline version. Several improvements were made to improve the performance on GPUs:
  - Overallocating the memory buffers on GPU to avoid allocation and deallocation on every simulation step proved to improve the performance of the kernel by about 4.5% in dynamic simulations.

  - Reducing the floating point precision from double-precision to single-precision lead to an additional increase of about 200%, due to the faster arithmetic logic units that are available on the GPU for single-precision floating point operations.
  - Performing sorting on the simulation objects' state data based on a space-filling curve provided an additional 2.5×speedup as it makes it possible for the GPU to enjoy spatial and temporal locality of cached data, and effectively reducing data-starvation of the computation pipelines.
  - Redesigning the kernel to enable use of shared memory resources on the GPU turned out worsen the overall performance, as the operations involved in creating the shared data objects need to be done in serial, and therefore reduces the parallelizable part of the kernel code. We however proposed a solution that extends on the idea of utilizing shared memory resources, and could potentially improve the performance further than currently possible.

- FPGAs are becoming increasingly more programmable for use in software development, and therefore we decided to investigate its potential by writing an OpenCL kernel that can be compiled to Verilog for high-level synthesis on the FPGA. The initial FPGA kernel version performed worse than the serial baseline version due the fact that we used the initial version of the GPU kernel to start with, and was therefore in accordance to our expectation. Several efforts were made to improve the performance:

  - The use of a look-up table (LUT) for one of the most repeated operation in the mechanical interactions module improved the performance by an insignificant percentage, but did decrease the number of FPGA resources required for the kernel. The impact of this improvement would be seen at a later stage, when the duplication of resources for performance improvement was implemented.
  - Loop unrolling in OpenCL for FPGAs makes it possible to duplicate the hardware needed for the kernel to be executed, and increases the pipeline parallelism that makes FPGAs unique in their architecture. We witnessed a close to 3×speedup compared to the previous version of the FPGA kernel, which translates to a 2×in comparison to the serial baseline version. From the Intel OpenCL for FPGAs SDK profiling tool we found out that the computational pipeline is data-starved due to the fact that memory accesses to the simulation objects' state data were not strided. This disabled the Intel Offline Compiler to generate hardware for coalesced memory accesses, and thus reduced the overall throughput of data.
  - In order to enable the generation of hardware for coalesced memory accessing, we redesigned the data structures such that a strided access pattern would be possible on the FPGA. To that extent we moved the computations required for the UG method to the CPU, such that the FPGA only performs the computations required for the mechanical interactions. This lead to a significant reduction in execution time on the FPGA, but the overall speedup was nearly nullified by the time spent on preparing the data structures on the CPU.

- Initially required for more realistic simulation, the extracellular diffusion module was another contribution of this work. It enabled us to identify the computation bottlenecks that arise from the simulations that can currently be run with BioDynaMo. The partial differential equation is solved on the CPU through a finite difference method, and fully parallelized with OpenMP. The additional API to initialize the extracellular substances with any user-defined funcion was also part of this work.

- The visualization of BioDynaMo with ParaView was another contribution of this work.

- We partially implemented the backup and restore functionality of BioDynaMo to allow long-running simulations to recover from unexpected failures. Several of the modules that belong to CERN's data processing and analysis software, ROOT, were utilized to this extent.

As a result, and to the best of our knowledge, BioDynaMo is currently the first agent-based biological simulation platform that makes use of hardware acceleration.

6.2. Future Work

This work has tackled one of the main obstacles that prevented BioDynaMo from simulating complex and large-scale biological models. As a result, the performance profile of BioDynaMo simulations has become flatter, and gives rise to other core modules in BioDynaMo to become the next computation bottleneck.

6.2.1. GPU kernel for extracellular diffusion

In recent efforts by our peers from the University of Newcastle, the extracellular diffusion can be very time consuming in the presence of multiple extracellular substances, and a high-resolution diffusion grid. The code for the finite different method, that solves the partial different equation of diffusion, is currently parallelized through OpenMP. Porting the code to the GPU as a CUDA or OpenCL kernel should be done in order to solve the computations for high-resolution diffusion grids in a shorter time span. An interesting investigation would be to use the GPU-offloading capabilities of recent version of OpenMP, and compare the performance against a CUDA or OpenCL kernel. We expect the performance of OpenMP to be lower than a custom kernel, but the development time could be significantly less, which would make OpenMP a feasible solution for future development of parallel code in BioDynaMo.

6.2.2. Distributed BioDynaMo

In recent efforts by our peers at CERN openlab, the distributed runtime of BioDynaMo could make it possible to not only *scale up* the simulations on a single compute node, but also to *scale out* to a network of nodes. The general concept is illustrated in Figure 6.1. The simulation space can be distributed among multiple compute nodes, where each node is responsible for the computations that is required for its subspace. Any boundary information that is required by neighboring nodes should be exchanged between nodes. Due to the local nature of agent-based simulations, the information of simulation objects that are not in the local vicinity is not required, and therefore the exchange of data needs to happen only between nodes that own the subspaces surrounding each other.

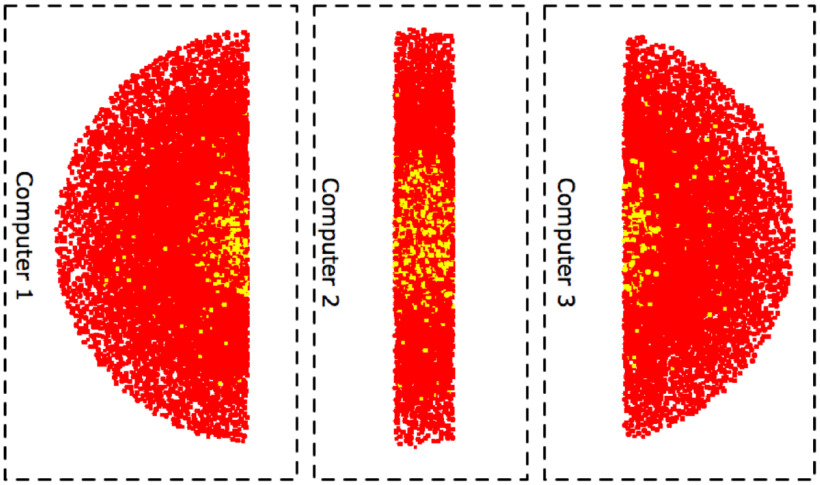


Figure 6.1: Distribution of simulation space over multiple compute nodes.

6.2.3. Heterogeneous Distributed BioDynaMo

In order to connect the work done in this work with the future distributed runtime of BioDynaMo, the follow-up work could be a heterogeneous distributed runtime. In such a setup some, or possible all, of the compute nodes could be equipped with on or more hardware accelerators that could be used in some of computation on the subspace that they host. There are different methods to integrate hardware accelerators in a distributed runtime, of which hardware virtualization appears to be the most suitable method, due to its natural compatibility with cloud computing resources, which are mostly virtualized resources. We have investigated several virtualization techniques in order to kick-start the work that would go into realizing a heterogeneous distributed runtime of BioDynaMo. In specific we

shall focus on GPU virtualization.

There is currently a lot of research going into providing virtualized distributed environments, where hardware accelerators such as GPUs are being virtualized [22]. The main goal of GPU virtualization is to utilize hardware accelerators more efficiently, by enabling resource sharing among multiple users or applications. Cloud providers are the main target group, because they start to offer increasingly more instances with GPU hardware available. GPUs consume significant amounts of energy when they are idle, which is wasteful. GPU virtualization could reduce the amount of wasted energy by sharing the resources with multiple users. Another goal of GPU virtualization is simplifying the way to create a heterogeneous distributed environment, by abstracting most of the underlying API calls to GPU programming frameworks, such as CUDA and OpenCL. That is why we have investigated the possibility to build the distributed runtime of BioDynaMo on top of an existing virtualization framework, and improve it for better performance for our use case. The authors of [22] have categorized most of the existing GPU virtualization solutions into three types: 1) API remoting, 2) full virtualization, and 3) hardware-supported virtualization. We will give a brief summary of each type.

API remoting
In API remoting the idea is that the computing resources (CPUs, GPUs) can be used by an application as though they were all locally available (i.e. on a single machine), while in fact an abstraction layer takes care of the actual inter-node communication. For instance, we could launch a GPU kernel on a machine that has no GPU hardware, but the framework would take care of allocating and copying the necessary memory buffers and launching the kernel on an available remote machine with a GPU.

Full virtualization
In full virtualization the GPU hardware is fully emulated by a hypervisor through virtualization of the GPU drivers. This allows users to access the GPU through a virtualized (guest) operating system, that can be different from the host operating system. Solutions exists where some of the operations are directly handled by the host driver, rather than the guest driver.

Hardware-supported virtualization
In hardware-supported virtualization the guest operating system is able to directly access the GPU through PCI pass-through. With PCI pass-through the OS which tries to access the GPU gets full control of the device, while the other OS is blocked from accessing the device. In recent development by NVidia (NVidia GRID [23]), it has been made possible to share a single GPU with multiple OSs by multiplexing the kernels that are queued up for execution on a GPU.

Bibliography

[1] NVIDIA Corporation, *NVIDIA Fermi Compute Architecture White Paper,* (2010).

[2] D. Eppstein, M. T. Goodrich, and J. Z. Sun, *The skip quadtree: a simple dynamic data structure for multidimensional data,* in *Proceedings of the twenty-first annual symposium on Computational geometry* (ACM, 2005) pp. 296–305.

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

[4] A. Bracciali and E. Larsson, *Data-intensive modelling and simulation in life sciences and socio-economical and physical sciences,* Data Science and Engineering **2**, 197 (2017).

[5] A. Hauri, *Self-construction in the Context of Cortical Growth: From One Cell to a Cortex to a Programming Paradigm for Self-constructing Systems*, Ph.D. thesis, ETH (2013).

[6] L. Breitwieser, R. Bauer, A. Di Meglio, L. Johard, M. Kaiser, M. Manca, M. Mazzara, F. Rademakers, and M. Talanov, *The biodynamo project: Creating a platform for large-scale reproducible biological simulations,* arXiv preprint arXiv:1608.04967 (2016).

[7] *Biodynamo,* `https://github.com/BioDynaMo/biodynamo` (2018).

[8] F. Zubler, A. Hauri, S. Pfister, R. Bauer, J. C. Anderson, A. M. Whatley, and R. J. Douglas, *Simulating cortical development as a self constructing process: A novel multi-scale approach combining molecular and physical aspects,* PLOS Computational Biology **9**, 1 (2013).

[9] I. Present, *Cramming more components onto integrated circuits,* Readings in computer architecture **56** (2000).

[10] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs* (Newnes, 2012).

[11] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, *A reconfigurable fabric for accelerating large-scale datacenter services,* ACM SIGARCH Computer Architecture News **42**, 13 (2014).

[12] Z. Wang, J. D. Butner, R. Kerketta, V. Cristini, and T. S. Deisboeck, *Simulating cancer growth with multiscale agent-based modeling,* in *Seminars in cancer biology*, Vol. 30 (Elsevier, 2015) pp. 70–78.

[13] 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 **9**, e1002970 (2013).

[14] Q. Wu, C. Yang, T. Tang, and L. Xiao, *Exploiting hierarchy parallelism for molecular dynamics on a petascale heterogeneous system,* Journal of Parallel and Distributed Computing **73**, 1592 (2013), heterogeneity in Parallel and Distributed Computing.

[15] NVIDIA Corporation, *NVIDIA CUDA C programming guide,* (2018), version 9.2.

[16] R. Brun and F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework,* Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997).

[17] A. Hesam and F. Rademakers, *Integrating ROOT I/O in BioDynaMo Brain Simulation Platform,* (2016).

[18] P. G. de Aledo, A. Vladimirov, M. Manca, J. Baugh, R. Asai, M. Kaiser, and R. Bauer, *An optimization approach for agent-based computational models of biological development,* Advances in Engineering Software **121**, 262 (2018).

[19] S. Green, *Particle simulation using cuda,* NVIDIA whitepaper **6**, 121 (2010).

[20] Intel Corporation, *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide,* (2018).

[21] NVIDIA Corporation, *NVIDIA GeForce GTX 1080 White Paper,* (2016).

[22] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, *Gpu virtualization and scheduling methods: A comprehensive survey,* ACM Comput. Surv. **50**, 35:1 (2017).

[23] A. Herrera, *Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation,* Nvidia Corp (2014).

A

```
// For all time steps
for (int t = 0; t < T; t++) {
    // For all cells
    for (CellType cell : cells) {
        // For each neighbor in cell's local environment
        for (CellType neighbor : cell.GetLocalEnvironment()) {
            double r1 = cell.GetRadius();
            double r2 = neighbour.GetRadius();
            double3 p1 = cell.GetPosition();
            double3 p2 = neighbour.GetPosition();

            // A result variable of type double3 for each cell (non-local scope)
            result = {0, 0, 0};

            double delta = r1 + r2 - length(p1 - p2);
            if (delta < 0) { continue; }
            // kEpsilon is defined by the modeler
            if (length(p1 - p2) < kEpsilon) {
              result = GetRandomDouble3();
              continue;
            }

            double r = (r1 * r2) / (r1 + r2);
            // kKappa and kGamma are defined by the modeler
            double f = kKappa * delta - kGamma * sqrt(r * delta);
            result = (f / length(p1 - p2)) * (p1 - p2);
        }
    }

    // Apply the result for all cells
    for (CellType cell : cells) {
        ApplyChanges(cell, result)
    }
}
```

Listing 4: Pseudocode for the mechanical interaction in BioDynaMo