The application of GPU to molecular communication studies

Tobias J. Cain
Eastern Washington University

Follow this and additional works at: https://dc.ewu.edu/theses

Part of the Graphics and Human Computer Interfaces Commons, Numerical Analysis and Scientific Computing Commons, and the Software Engineering Commons

Recommended Citation
https://dc.ewu.edu/theses/488

This Thesis is brought to you for free and open access by the Student Research and Creative Works at EWU Digital Commons. It has been accepted for inclusion in EWU Masters Thesis Collection by an authorized administrator of EWU Digital Commons. For more information, please contact jotto@ewu.edu.
THE APPLICATION OF GPU TO MOLECULAR COMMUNICATION STUDIES

A Thesis
Presented To
Eastern Washington University
Cheney, Washington

In Partial Fulfillment of the Requirements
For the Degree
Master of Science in Computer Science

By
Tobias J. Cain
Spring 2018
THESIS OF TOBIAS CAIN APPROVED BY

DATE
DR. YUN TIAN, GRADUATE STUDY COMMITTEE

DATE
DR. URI ROGERS, GRADUATE STUDY COMMITTEE
Abstract

This thesis applies the recent trends in parallel processing, via graphics processing unit (GPU), to the field of molecular communications (MC), an investigation into communication possibilities of futuristic in vivo nanomachines. Existing MC simulations have not fully accounted for structural boundaries and the associated simulation of a massive number of messenger molecule paths for stochastic evaluation. These molecules are influenced by a Brownian motion as well as the flow of the blood, which is modeled using numerical methods based on the Fokker-Planck stochastic differential equation. By using a GPU these paths can be calculated on a massive scale, both in the number of simulated paths, and the number of time steps. The use of a GPU also allows for other obstacles and complications to be added to the path of those molecules in future works. This study should enable and expedite existing as well as future study in the MC field.
Acknowledgement

A special thank you to Dr. Yun Tian and Dr. Uri Rogers. You have both been an incredible help through this process. Your guidance and extensive knowledge of your respective fields has been invaluable. I would also like to thank Dr. Andrew Oster for acting as my third committee member.
## Contents

1 Introduction ........................................... 1  

2 Related Works ...................................... 3  
   2.1 Molecular Communication ........................................... 3  
   2.2 Graphics Processing Unit Background ........................................... 5  
   2.3 Random number generators on GPU ........................................... 6  

3 Methodology ........................................... 9  
   3.1 Molecule Simulation ........................................... 9  
   3.2 Data storage ........................................... 15  
   3.3 Options ........................................... 15  

4 Evaluation ........................................... 17  
   4.1 Validation ........................................... 17  
   4.2 Speed Comparison ........................................... 19  
   4.3 Robustness Comparison ........................................... 24  
   4.4 Reflecting Walls ........................................... 25  

5 Discussion ........................................... 26  

6 Future Work ........................................... 28  

7 Conclusion ........................................... 29
List of Figures

1.1 Conceptual Molecular Communication Distributed Detection system [1] .......... 1
2.1 Block Diagram of the GP104 GPU [2] ................................................. 5
2.2 GP104 Streaming Multiprocessor Diagram [2] ................................. 7
2.3 Monte Carlo simulation used to estimate π [3] ................................. 8
2.4 SP network consisting of narrow S-boxes, which are bijections. The S-boxes then feed into a P-box and are shuffled. [4] ................................. 10
3.1 The autocorrelation of C function compared with MATLAB. ................ 12
3.2 The autocorrelation of CUDA function compared with MATLAB. .......... 13
3.3 Wide implementation on GPU ......................................................... 13
3.4 Long implementation on GPU ......................................................... 14
3.5 An example of the help menu ......................................................... 16
4.1 Hit probability for a single receiver, without drift, at d = 50nm from transmitter. 10E6 paths calculated ................................. 18
4.2 Hit probability for a single receiver, with drift, at b = 300nm from transmitter in R^1 19
4.3 The path of a single particle without drift ......................................... 20
4.4 Speed comparison with each step recorded ....................................... 21
4.5 Speed comparison for R^3 test, without drift .................................... 22
4.6 Speed multiplier per number of paths run ....................................... 23
4.7 Reflections of 8µm tube ................................................................. 26

List of Tables

4.1 First-hit CPU vs. GPU-W vs. GPU-L Speed Comparisons for R^1 Planar Test with z-limit at 3E-7 and Drift enabled ........................................ 22
4.2 CPU vs. GPU-L, 1E4 iterations, 1E-3 end time, first hit, no drift ............... 24
4.3 CPU vs. GPU-L, 1E4 iterations, 1E-3 end time, first hit, no drift ............... 25
5.1 First-hit GPU-W vs. GPU-L Speed Comparisons for R^3 Receiver Test ........ 28
1 Introduction

Nanotechnology is a new and emerging field, and has been studied in recent years as a new way of interacting with the body, *in vivo* [5]. Nanotechnologies have the possibility of delivering drugs to exact areas in the body, acting as *in vivo* imaging probes or providing early stage disease detection [6]. However, many of the capabilities hinge on the existence of a suitable network. Networks currently available do not meet the size or power requirement needed to exist within the human body. These networks generally fall into one of two classifications, wired or wireless. Wired technologies use a sizeable amount of power as well as presenting an installation problem, while wireless technologies require antenna sizes that are generally too large to implement *in vivo* [6]. When considering geometric limitations and resource constraints, one can look to nature for alternative communication methods. Molecular communication focuses on utilizing molecules to communicate, similar in form to the communication patterns platelets utilize in blood [7]. Both passive and active transportation techniques are currently being studied [5, 6]. As the molecules are transported through the body, the flow and contents of the blood affect the position and path of those particles. Red blood cells have a significant effect on the bloodstream due to their relative size in comparison with other particles [8]. One possible example of molecular communication described by Rogers *et al.* is shown in Figure 1.1 [1]. Where nanomachines that detect a disease biomarker
release a \( \beta \) molecule, while nanomachines that do not detect said biomarker, release an \( \alpha \) molecule to be detected by a fusion center. The fusion center will then decide whether a disease is present or not based upon the received responses. Cursory research models explore molecular movement; however, they are not suitable for statistical analysis needed for MC due to limitations in speed and robustness. In a previous paper we have shown that with multiple receivers, non-linear spatial dependency can be found [1]. These findings emphasize the challenge in dealing with particles in the bloodstream and the complication in predicting where the particles will be or how likely they are to arrive at a receiving nanomachine.

When dealing with radio communication packets, multiple bits of data can be sent and received. While messenger molecules can be designed to transmit different bits, without loss of generality, this work focuses on binary messenger molecules. Each of these molecules has a distinct path, meaning that even if multiple molecules are released from the same location at the same time, the likelihood of any of those molecules reaching the receiver drop exponentially as the distance to the receiver increases.

Originating from the application of computer graphics, GPUs have since been used for massively parallel processes in many other fields. They are excellent at processing a massive number of similar commands; however, they are not ideal for general purpose computing [9]. Due to the independent attributes of the problem or the so-called "embarrassingly parallel" nature [10], a GPU is a good fit. MC requires an understanding of how molecules will move in the bloodstream; these movements rely on stochastic processes. In order for the data in this model to be as accurate as possible, a massive number of paths need to be numerically simulated with the most precise time step possible. This entails a tremendous number of calculations, on the scale of \( 10^6 \) and above. In a study of the location of a molecule, each simulated molecule is assumed to be independent of each other molecule, if the collision of released molecules is not considered. This allows each molecular time step to be calculated independent of each other molecule, best utilizing the parallel ability of the GPU. By calculating one step at a time, thousands of molecular paths can be calculated in parallel. This paper discusses the simulation of these particles via GPU while also looking to validate the results and increase the simulation speed relative to the CPU.

This paper is organized in the following manner. Section 2 will explore existing studies in molecular communication and look at ways GPUs have been implemented in other fields. Section 3
will discuss the implementation of molecular simulation on the GPU and design decisions. Section 4 will analyze the results achieved and verify their accuracy, while also looking at the time-cost improvements and added functionality. Section 5 will be a discussion of the findings noted in Section 4, taking a closer look at the newly available data. Section 6 will discuss future work and other possible applications. Section 7 will contain closing remarks and observations.

2 Related Works

This paper focuses on the application of a GPU to the simulation of particle movement for use in the MC field. This section will first discuss the background of MC and its application, then delve into the history and applied uses of the GPU.

2.1 Molecular Communication

The idea of nano-technologies in the body has long existed in the stories of science fiction. Recently, these fantasies have been approaching reality. Nanodevices currently exist in medicine; however, in most of these implementations, the devices function independently from each other in the bloodstream. While currently available nanomachines can do things such as recover from glucose feedback loops using nanopores, the complexity of these devices is limited due to power, size, and communication restrictions [6]. To expand the functionality of devices in the bloodstream, many communication methods have been explored, such as molecular, nanomechanical, electromagnetic and acoustic communications [6]. Among these, molecular and acoustic communication appear to be most bio-compatible. This paper will focus on molecular communication due to its similarity with natural bodily processes and the current availability of molecular communication equipment [6].

Molecular communication uses naturally existing molecules to communicate messages between transmitting and receiving nanomachines. This concept is based on a naturally occurring bodily processes. For example, white blood cells called B-cells will communicate with T-cells, via MC, to remove pathogens entering the body [11]. There are many ways in vivo MC has been approached, but most can be classified as either a passive or active transport. Active transport can utilize routes such as a molecular rail or microtubules [12]. While this, in theory, enables very consistent communication, the implementation and scalability remain a challenge. Passive transport relies on the use
of naturally existing forces to connect the two media. Within the bloodstream, there are two main forces, among many others, that act upon a molecule; Brownian motion and blood flow. Brownian motion’s effect on the molecules in a fluid is a known equation. This perceived movement that is Brownian motion is caused by the collision of other, smaller particles with the molecules. Blood flow can be divided up into two different areas when viewed from a high level; a core region where most cells are found and a cell-free layer [8]. A turbulent flow in the core region is caused by the tumbling of Red Blood Cells (RBCs), which are magnitudes larger than most other particles in the blood. In Tan et al., a model for the tracking and influence of RBC on nanoparticles (NP) is discussed using mean square displacement [8]. The turbulent flow pushes the smaller particles to the edges of the blood vessel, in an area called the cell-free layer, where they travel in a smoother flow referred to as laminar flow. Red blood cells are considered out of the scope of this paper, although they are important and should be considered in future works. This software architecture has been designed with these possible additions in mind. Laminar flow provides an ideal stream for passive MC. The movement of a given particle in the cell-free layer is, thus affected by Brownian motion with a laminar flow.

In order to model the individual molecular paths, it is important to understand how a molecule behaves. The Brownian motion can be simulated using the forward Kolmogorov (Fokker-Planck) equation

\[
\frac{\partial p_B(t, \xi; \xi_0, t_0)}{\partial t} = \frac{1}{2} \sum_{j=1}^{3} \sum_{k=1}^{3} \frac{\partial^2 D_{j,k} p_B}{\partial \xi_j \partial \xi_k} - \sum_{j=1}^{3} \frac{\partial v_j p_B}{\partial \xi_j}
\]

(2.1)

where \(D_{j,k}\) is a diffusion constant, \(v_j\) is the drift velocity in \(j\) direction, and \(p_B(t, \xi; t_0, \xi_0)\) is a pdf for emission location \(\xi\) at time \(t\) with starting points of \(\xi_0\) and \(t_0\) respectively [1]. The first portion of the equation can be thought of as the random motion in three dimensions with the second part of the equation accounting for the drift from the flow of blood. For this paper, the laminar flow is assumed to be in a singular direction, which is arbitrarily selected as \(j = 3\), or the \(z\)-axis. The flow is also assumed to have a constant velocity, although in the future works the application of Hagen-Poiseuille equation, which can be derived from the Navier-Stokes equation, may be applied. The Hagen-Poiseuille velocity distribution for laminar flow in a long tube is parabolic [13, p.48].
2.2 Graphics Processing Unit Background

Today the GPU is applied to many unique fields in order to provide speedups for inherently parallel problems, but originally the GPU was mainly focused on graphics processing. This trend continued until around 2007, when Nvidia announced a beta version of CUDA, a C based programming language, that could be used to harness the parallel nature of its video cards. This alternative use of the GPU became known as general purpose GPU usage, or GPGPU [9]. While the new technological advancement was exciting, GPUs work differently than the traditional CPU, so adjustments needed to be made. While the CPU is focused on general purpose tasks needed for everyday use, the GPU, as shown in Figure 2.1, is made up of many streaming microprocessors (SMs), shown in Figure 2.2. These SMs are highly parallel multiprocessors that schedule groups of 32 threads, called warps, to the CUDA cores [2]. Each SM contains some number of CUDA or scalar cores, in the case of

Figure 2.1: Block Diagram of the GP104 GPU [2]
Figure 2.2, 128 CUDA cores are on each SM. It is not uncommon for consumer CPUs to have 2-8 cores, compare this to the number in the Nvidia GeForce GTX 1070, a mid-range consumer GPU, at 1920 CUDA cores. It then makes sense that CPUs focus on minimizing latency while GPUs focus on maximizing throughput. CUDA programs are based around kernels, which are function calls, highly parallel in nature, that create thousands of threads on the GPU after invocation. These threads are grouped into blocks and ultimately a grid. Each of these threads run an identical kernel function, independent of each other in both result and completion order. Memory exists on both the host (CPU) and device (GPU) side and must be transferred back and forth from host and device in order to be accessed by the respective sides. This means that data must be transferred to device memory in order to be processed by the GPU. Subsequently, it must be transferred back to host memory in order to be referenced by the CPU. Thus, for highly parallelizable or so-called embarrassingly parallel problems, the GPU is commonly superior to the CPU in processing time. These speed inequalities are only exacerbated as the number of identical tasks increase. Though for a small number of parallel tasks, the CPU is generally faster. This is partially due to the required memory transfer between the host and device. In the paper GPU Computing, Owens et al., shows that for a task of scanning elements, a CPU is faster at scanning elements when there are $10^4$ or fewer, but is quickly surpassed as the number of elements increases [9]. The GPU has been used in many different fields including applications in stochastic differential equations [10].

2.3 Random number generators on GPU

One of the issues Monte Carlo simulations or other stochastic based problems traditionally face is the reliance on a random number generators (RNG). These simulations are generated using a massive number of random values. One typical example of a Monte Carlo application is using randomly generated numbers to determine $\pi$ [3]. This can be done by enclosing a circle within a square, as seen in Figure 2.3. A circle with known diameter of $d = 2r$, will relate to the side of a square, $a$, in the form $a = 2r$. Thus, the areas can be defined as $\pi r^2$ and $4r^2$ respectively. Comparing the areas as a ratio, $p$, we are presented with

$$p = \frac{\pi r^2}{4r^2}.$$
With a little manipulation, \( \pi = 4p \) can be attained. By plotting random points then considering the ratio of the points that occur inside versus outside of the circle, an increasingly accurate approximation of \( \pi \) can be achieved as \( n \to \infty \), where \( n \) is the number of random points plotted [3].
Another example application of Monte Carlo simulations is Brownian motion [3]. Thus, the need for an efficient RNG. Extensive research has been put into the application of RNGs on the CPU; however, the research is done with the CPU architecture in mind, as such they are inherently sequential [3]. There are significant differences in the architecture of a CPU when compared to that of a GPU, hence a RNG must be designed with the GPU in mind. There are two factors to consider when selecting a RNG for use on a GPU: the speed and its randomness. These are generally inversely proportional properties. For speed considerations, true RNGs are eliminated from consideration and Pseudo-Random Number Generators (PRNG) are considered instead. While a true RNG may be irreplaceable in applications such as online poker, a PRNG can be perfectly adequate for other simulations [14]. For both CPU and GPU implementations a PRNG is used in this project. For this application the consideration of randomness versus speed was equally important. The PRNGs are a little different than a true RNG, in that they typically rely on a seed that is used to initialize a state that will follow a specific pattern. Each seed is guaranteed to produce the same state and sequence, this means that the same seed should not be used twice [4]. The change in the seed, also means that the state must be re-initialized. This is a potentially time consuming task and
must be considered. Several PRNGs were tested with these considerations in mind. The default PRNG used by CUDA is XORWOW [15]. It is very fast at the generation of the random numbers, but has a very costly initialization time for the state [15]. Another PRNG that was tested in place of XORWOW for this implementation was MRG32k3a, a member of the combined multiple recursive PRNG family, it was found to be slightly slower in both generation and state initialization [15]. The PRNG that was chosen is the Philox model [4]. This model had an inexpensive state initialization and was found to be very efficient overall while producing random numbers with little correlation. A software library, known as TestU01, has been used to test the Philox PRNG [4]. Three different sizes of tests are available in this package, SmallCrush, Crush, and BigCrush. BigCrush contains 106 tests and will take a few hours to run on a modern CPU [4]. If a PRNG passes the extensive empirical validation batteries with zero failures it is said to be Crush-resistant [4]. Philox is a counter-based PRNG, meaning that the state is kept by an integer, instead of a state transition function [4]. This reduces the cost in the state initialization. Philox is an SP network, referred to as such due to its construction of substitution and permutation blocks as discussed in [16]. The structure can be seen in Figure 2.4, which is described by Salmon et al. as, "a highly diffusive bijection . . . obtained by iterating successive rounds of substitution, called S-boxes (narrow bijections, applied in parallel), and permutations, called P-boxes (wider, but highly structured mappings that simply permute the bits in a p-bit space)." [4, p.10]. It has been tested to be Crush-resistant and found to be the highest overall throughput for a PRNG, at the time the paper was written, at 202GB per second on a NVidia GTX580 [4]. The PRNG used on the CPU is using the Marsaglia and Bray method [17], potential cost savings could be found by utilizing Threefish, a multi-core counter-based PRNG [4]. However, multi-core implementation of the CPU is outside the purview of this project.

3 Methodology

3.1 Molecule Simulation

As a entry point for construction of a simulation, the forces that affect the movement of the molecule must be determined. This has been discussed in our previous paper [1]. Particle movement is based off of a Brownian motion and can be represented as stated in equation 2.1. Laminar drift can be assumed, by either ignoring RBCs in the short term or assuming particle movement is limited to the cell-free layer of blood vessels. The tumbling motion of the RBC in the bloodstream forces most
smaller molecules, such as the proposed nanomachine into this layer \[8\]. Assuming distributed
detection, a passive form of molecular communication is used, the released molecules will need
to be detected downstream in order to communicate. This means the molecules released upstream
will have one of two possible routes from the view of the receiver. Either the molecule will reach
the receiver, or it will miss. If a molecule takes longer than a set amount of time, the molecule
is also considered to have missed. This simulation has been performed in \[5, 18\] by Rogers \textit{et
al.}, and was used as a baseline for this research. For this simulation, a singular starting point at
some \(\mathbf{x}_0(x_0, y_0, z_0)\), and a static receiver are assumed. This means that each molecule will behave
according to a Brownian motion, and will be tracked for a chosen period of time, \(T\). If at any point
that molecule enters the receiver, the first-hit time is recorded, no subsequent times are recorded.
If by time, \(T\), the molecule has not been detected, then it is assumed to have missed. With a static
diffusion rate, the output of the simulation can be compared and validated using analytical solutions
\[19, 20\]. The solutions have been implemented on both the CPU and GPU for comparison purposes.
In order to implement the algorithm for Brownian motion was adapted to,
\[
\sqrt{2 \ast D_b \ast \Delta t \ast \text{randn}},
\]
where \(D_b\) is the diffusion factor, \(\Delta t\) is the time step size and \(\text{randn}\) represents a random normalized
value. This equation is used for the \(x\), \(y\) and \(z\) directions, when laminar drift is assumed absent. This
equation represents the path of a single molecule to an end time, \(T_{\text{end}}\). To gain understanding of

Figure 2.4: SP network consisting of narrow S-boxes, which are bijections. The S-boxes then feed
into a P-box and are shuffled. \[4\]
the probability of a single particle hitting the receiver, this simulation must be repeated a significant number of times and the statistics calculated across each simulated sample path. The CPU code is written in C++, and the GPU code is C++/CUDA. The CPU was implemented using an $\Theta(N^2)$ algorithm,

**Algorithm 1: CPU implementation of Brownian Motion**

```plaintext
for $P < \text{number of paths}$ do
  for $t_s < \text{number of time steps}$ do
    rand$_x$ = randn$_h$ost; rand$_y$ = randn$_h$ost; rand$_z$ = randn$_h$ost;
    $x_{next}$ += $\sqrt{2} \cdot Db \cdot \Delta t \cdot \text{rand}_x$;
    $y_{next}$ += $\sqrt{2} \cdot Db \cdot \Delta t \cdot \text{rand}_y$;
    $z_{next}$ += $\sqrt{2} \cdot Db \cdot \Delta t \cdot \text{rand}_z$;
  end
end
```

A PRNG is required for this equation. The built-in PRNG was not suitable for our needs due to the lack of a Normal distribution. In order to get the PRNG needed the Marsaglia and Bray also known as the Polar method was used [17]. The normalized pseudo-random number generator results were determined usable via autocorrelation in comparison to the randn output of MATLAB. As can be seen from Figure 3.1, there is very little correlation between the results and the expected value.

The CUDA code uses the Philox PRNG as discussed in Section 2. While it has been crush tested, the same autocorrelation validation was used to provide a relation between the two methods. It can again be seen the correlation is low and is acceptable for the needs of this paper.

In moving the equation from a sequential to a parallel structure, a different approach required. While the CPU is traditionally based on a single instruction stream single data stream (SISD) structure, the GPU is best utilized when using single instruction stream, multiple data streams (SIMD) structure. For the molecular movement via Brownian motion, each step is dependent on the last in the form

$$x_{\text{new}} = x_{\text{last}} + x_{\text{next}},$$

where $x_{\text{new}}$ is found by adding the previous $x$ coordinate, $x_{\text{last}}$, to the newly calculated movement,
Depending on the implementation used, this could be an issue for the GPU. The GPU is able to calculate a large number of items in parallel; however, they are not guaranteed to be started or completed in any given order. This means that if the same implementation were used, the movement of the molecule would not be guaranteed to move in chronological order. A case exists in which, $t_0$ could come after $t_n$, where $n > 0$. Two approaches were considered, the first calculates $n$ paths one step at a time, each step being computed by one kernel launch. This, in turn, creates $m$ threads, each thread processing one step of one unique molecule. This first approach will be referred to as the wide implementation.

A path can be defined as the route a molecule takes in a given time period, $t$. An example path can be seen in Figure 4.3. This means that if $n$ paths were to be calculated, for each path the position value at $t_0$ would be found, as seen in Figure 3.3. Once all calculations for $t_0$ had been determined, $t_1$ is then calculated in the same way $t_0$ was, continuing this pattern to $t_n$. By approaching the calculation in this manner, the output can be passed forward each time without any reliance on past data, whilst still taking advantage of the parallel structure required for the GPU. This inherently
implies that the GPU algorithm will be better at calculating a large number of paths than it will paths that are followed for a long period of time. This differs from the CPU implementation as the CPU calculates an entire path to $t_{end}$, then moves on to the next path. Thus, each path is calculated in its entirety, one after another on the CPU.

The next route explored focused on better utilizing the computational power of each thread; this will be referred to as the long implementation. The long design was similar to the CPU design in that each thread calculated a path from $t_0$ to $t_{end}$. They differed in the fact that the CPU only
Figure 3.4: Long implementation on GPU

calculated a single path at a time, while the GPU would run \( n \) threads in parallel upon a kernel launch. This implementation gave each thread the responsibility of an entire path along with any logic that came with that. This approach applied more stress to each thread but removed much of the start-up cost of the threads due to the reduced number of kernel calls. It also made better use of the RNG, as each seed was used \( s \) times, where \( s \) is the number of time steps taken. This differs from the wide approach of a single use per seed.

The next thing to be added to the simulation was the effect of laminar flow (aka Brownian drift). The force is assumed to be applied in a singular \( z \)-direction. By applying a constant velocity in the direction of \( z \) at each time step, we can simulate laminar drift. This makes the full equation for the \( z \)-direction,

\[
z_{\text{next}} = \sqrt{2 \ast D_b \ast \Delta t \ast \text{rand}_z} + \text{vel} \ast \Delta t,
\]

where the velocity is a constant that is applied every \( \Delta t \).

Conditions and reflection properties needed to be added to both applications in order to be able to utilize the data that is being represented. In both implementations the time step is cycled over, meaning that conditions, reflections or collisions can be looked at based on geometry and \( \text{if} \) statements. In Section 4, the tests will be discussed. The conditions needed to be established before performing these tests. The first looked to see if the particle reached a point of radius, \( r \). This could be represented,
if \((r^2 > (x - \text{rec}_x)^2 + (y - \text{rec}_y)^2 + (z - \text{rec}_z)^2)\),

where \text{rec}_x, \text{rec}_y and \text{rec}_z, are the 3-Dimensional coordinates of the receiver. This implementation works great for the GPU as it is a basic condition that can be looked at each loop with very little slowdown. Likewise, the planar test and basic collision logic can be realized in a similar form.

### 3.2 Data storage

Typically a method of data storage needs to be established. The structure must be efficient or simple due to the number of iterations it would be expected to work with. It also needed to be able to access points in a similar area quickly. The first route examined was a tree-structure of some type. Octrees and Kd-trees were both looked at and carefully considered early in the project. However, once it was decided that the data would be calculated in steps, it made sense to instead do real-time calculations for tests or when a collision was detected. Due to the known three-dimensional position of all molecules at a time, \(t_n\), obstacles, and reflections of particles due to added obstacles may be explored in the future and are discussed more in Section 6. As the number of collisions increases, it may still make sense to use a tree structure, especially if those collisions are particle to particle. A tree structure may also make sense if the data needs to be acted on after leaving the GPU. For the current design, an array structure is used in order to look ahead and behind by one step, which is needed by the Brownian motion algorithm as well as geometric condition tests and reflection algorithms.

### 3.3 Options

The program was coded in C/C++, with the GPU sections being coded in CUDA. It was designed as a command line program that outputs data to CSV files for analysis with the application of choice. The program uses command-line arguments to define the variables in the program. Many options are available and can be mixed and matched as needed for the research being done as seen in Figure 3.5. It has been designed to be a single run instance. For some of the timing used in this article, BASH scripts, a Linux based scripting that can be used from the terminal, have been used to loop through the execution of the program. This could be added if the use case was found to be needed outside of this paper.
The program has a variety of available outputs. It can output, every three-dimensional point for each time step allowing the graphing of a full molecular path. This is only available in the wide model. This is due to the long model’s design of using a thread for a full path simulation. There is less of a chance to access the data without large memory usage. Having the paths plotted aids in troubleshooting, as well as understanding how a path may look. However, as the number of paths increase, the data becomes massive, and more cumbersome to use. The saved data files quickly expand to many gigabytes and only increases from there. In order to reduce this issue, while still collecting important data, the option for a hit-time was added. This means that only the particles that make it to the set obstacle are recorded, be it a $\mathbb{R}^1$ plane or at a $\mathbb{R}^3$ receiver. At each time step the particle will be compared to the hit-area and recorded, if it is found to be inside. This means that for each particle that achieves the set condition it is possible, if not likely, to have several points recorded. A similar option, records only the first-hit time. This means that all time steps after the first hit are discarded. These concessions allow for significantly larger simulations to be run. To implement the collisions, the bounds for the specific objects are set within the code. For things such as the walls of the blood vessel, some parameters, in this case the radius of the blood vessel, are made available for the user to set. Detection tests have been designed for the hit-receiver and a planar limit. By implementing the collisions in this manner, the GPU can be utilized to perform these checks and reactions in real-time, instead of having to store and then re-access the data. Other obstacles may be added in the future and will be discussed in Section 6. The GPU blocksize,
Δt, planar limit, and simulation end time are all available to be changed from the command-line interface. A compare option also exists that can be used to compare the results of the GPU and the CPU. By default it will only run the GPU. Defaults are also set for most settings, and can be viewed from the help menu as well as any other available options, which can be accessed using -h or –help, as visible in Figure 3.5.

4 Evaluation

A goal of this thesis was to enable study that had proved difficult or unavailable at the time this paper was written due to speed limitations. In this pursuit, two main items had to be examined, the speed of the analysis and the accuracy of the findings. The speed at which the simulation operates is important in order to be able to run different experiments and compare the results quickly. The time in which a test can be completed has a direct effect on the number of tests run. Due to the stochastic nature of data produced, it is also vital to have a large sample size.

4.1 Validation

In order to compare the speed of the processes, it was important first to identify some benchmarks that could be compared on both the CPU and the GPU. It was also key that the algorithms used could be compared to and validated by previously vetted results. From this perspective, two main tests were implemented.

The first test implemented focuses on the paths of the particles in 3-dimensional space. Given a starting point, $\xi_0$, in $\mathbb{R}^3$ the test records the time each path takes to reach a receiver of set $r$ at a given $\xi_{end}$. The receiver has a radius, $a$, and exists a distance, $d$ from $\xi_0$. For each particle that reaches the receiver, the first-hit time is recorded. For this simulation, no walls, drift or other variables are considered. The particles move in the medium according to Brownian motion, and those that reach the receiver are recorded.

If the detecting surface is supposed to be perfectly absorbing, then the equation for the so-called reaction rate at $r = a$ for a receiver of radius $a$, can be written

$$H_{Diff}(t|d, t_0) = \frac{1}{d \sqrt{4\pi D(t-t_0)}} \frac{d-a}{t-t_0} \times \exp \left[ -\frac{(d-a)^2}{4D(t-t_0)} \right],$$  \hspace{1cm} (4.1)$$

where $t$ is time, $t_0$ is the start time, and $d$ is the distance of release point to the receiver [5]. The hit
probability for a pure diffusion process can then be calculated using,

\[ Pr(m_k \text{ hit } R) = \frac{a}{d}. \] (4.2)

This equation is justified by Schulten et al. by massaging a well-known diffusion equation, then taking the limit as \((t - t_0) \to \infty\), with the assumption of a perfectly absorbing surface [19, Sec.3.4].

![Figure 4.1: Hit probability for a single receiver, without drift, at \(d = 50\text{nm}\) from transmitter. 10E6 paths calculated](image)

As can be seen in Figure 4.1, the GPU simulation matches the analytical solution very well. Due to the time-step chosen, 1E-7, the graph is missing part of the first leg. If a smaller time step were chosen, the missing leg would be smaller.

The next test used for validation looks at the probability a particle will reach a point in \(\mathbb{R}^1\). This calculation is done in \(\mathbb{R}^1\) due to the lack of an analytical solution with drift in \(\mathbb{R}^3\). The particle begins at a point \(\xi_0\), at time \(t_0\) and will be recorded if it reaches a point, \(\xi_z\). Each tracked particle is acted on by Brownian motion, as well as a laminar drift in the \(z\)-direction. Therefore, with enough time each particle will reach the point \(\xi_z\). This means that the hit probability is \(Pr ( m_k \text{ hit } R) = 1\), the same cannot be said for channels where \(\frac{a}{d} << 1\) in \(\mathbb{R}^3\). The first-hitting time follows an inverse Gaussian distribution represented as
\begin{equation}
    p_{passage}(t) = \frac{b}{\sqrt{4\pi D_B t^3}} \exp\left(-\frac{(vt - b^2)}{4D_B t}\right),
\end{equation}

where \( b \) is the distance from the transmitter to the receiver, and \( v \) is the drift velocity, all in \( \mathbb{R}^1 \) [20, pg. 79].

![Graph showing hit probability for a single receiver, with drift, at \( b = 300nm \) from transmitter in \( \mathbb{R}^1 \)]

Figure 4.2: Hit probability for a single receiver, with drift, at \( b = 300nm \) from transmitter in \( \mathbb{R}^1 \)

The GPU output matches the analytical solution well. It is calculated using the first-hit time with 10E5 paths run. This means that only the first point to pass the \( z \) threshold was utilized for this test.

Examining these two tests, it is clear that the movement of the particle via Brownian motion and Laminar drift behave according to analytical results. The validation of these results allows data created by this program to be used in future studies as a verified starting point.

### 4.2 Speed Comparison

The testing was done on a Nvidia GeForce GTX 1070 and the CPU used was an Intel I7 Kaby Lake, Quad-Core 4.2GHz processor. The GTX 1070 uses the GP104 chip referenced in, Figure 2.1. It only has 15/20 SMs enabled, versus the GTX 1080 that has the full 20 enabled. The CPU is run
in a single core mode for the testing, it would likely increase the speed if the comparison algorithm were to be implemented using multiple cores with an alternate PRNG. Three chosen tests were run. The first was used as an eye test, to verify the output looked correct. This can be seen in Figure 4.3. It will be useful whenever a visual representation of the data is required. It can also be helpful when writing new functions as a way of quickly determining if the setup is working as expected when viewed from a high level.

In Figure 4.4, it can be seen that the CPU operates at a nearly linear rate for the entire test period, while the GPU sees a speed increase early then continues at a near linear speed. The limitation and similarity in speed is due to the need to write all data at each step. The write requires copying the data back from GPU memory to local memory. Then the CPU is used to write the data to a CSV file. The data does avoid having to be copied back from the host to the GPU and is instead altered from its previous value. The CPU version of this test must also write the data to a CSV; however, it writes a full path at a time versus each step as noted in Figure 3.3. The biggest reason the speeds are so close is due to the data write being performed on the CPU in both cases. This test was only able to be performed using the wide implementation of the GPU. The long version is unable to record the entirety of a path.

The next tests that were performed compared the speed increases for the calculation of the first
time found. Less than five percent difference was found between the runs with and without drift. This is logical as the drift only adds a single equation to the calculation. Initially in this test, the XORWOW PRNG was used and the initialization cost of the state, not only caused a massive slow down across all ranges, but after 2E5 paths, the GPU implementation actually became slower than the CPU implementation. For any larger number of paths the program would become unresponsive.

By utilizing a counter-based PRNG, a better speed is achieved, and the hard limit of 2E5 paths has been far extended. In Table 4.1, it can be seen that the wide implementation appears to plateau at a speedup of approximately 40x. The long setup has lightning fast speeds as can be seen in Figure 4.5, at 2E5 it does not even break 1s to compute. The differences and possible causes for this significant difference will be discussed in Section 5. It is curious that the time cost for the CPU is almost a direct multiple of both the Wide and Long GPU implementation costs as seen in Figure 4.5. However, this makes sense due to the lack of complex states that can exist within the implementation. This means that the longest possible time for a path to complete is very close to the minimal completion time for all cases. Thus, the speedup provided by the GPU is seen relatively equally across all paths calculated.

While the use or non-use of drift on the simulation has a minimal effect, the $\mathbb{R}^1$ planar test
Figure 4.5: Speed comparison for $\mathbb{R}^3$ test, without drift

Table 4.1: First-hit CPU vs. GPU-W vs. GPU-L Speed Comparisons for $\mathbb{R}^1$ Planar Test with z-limit at 3E-7 and Drift enabled

<table>
<thead>
<tr>
<th>Paths Simulated</th>
<th>CPU-Drift</th>
<th>GPU-Drift-Wide</th>
<th>Wide Speed-up</th>
<th>GPU-Drift-Long</th>
<th>Long Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E3</td>
<td>1.46s</td>
<td>0.28s</td>
<td>5.2x</td>
<td>0.006s</td>
<td>243x</td>
</tr>
<tr>
<td>5E3</td>
<td>7.47s</td>
<td>0.41s</td>
<td>18.2x</td>
<td>0.008s</td>
<td>934x</td>
</tr>
<tr>
<td>1E4</td>
<td>15.13s</td>
<td>0.61s</td>
<td>24.8x</td>
<td>0.012s</td>
<td>1260x</td>
</tr>
<tr>
<td>5E4</td>
<td>73.17s</td>
<td>1.95s</td>
<td>37.5x</td>
<td>0.055s</td>
<td>1330x</td>
</tr>
<tr>
<td>1E5</td>
<td>149.99s</td>
<td>3.81s</td>
<td>39.4x</td>
<td>0.12s</td>
<td>1250x</td>
</tr>
<tr>
<td>2E5</td>
<td>302.91s</td>
<td>8.26s</td>
<td>36.7x</td>
<td>0.23s</td>
<td>1317x</td>
</tr>
</tbody>
</table>

versus the $\mathbb{R}^3$ receiver test sees a large difference. In order to be able to plot entire molecular paths, the wide implementation transfers all data from the device (GPU) to the host (CPU) then applies the logic to it. Due to this reason, a very large difference in the speeds of the two tests can be observed.
The planar limit test has a much simpler test condition, a $\mathbb{R}^1$ threshold, that it compares to position each step. The difference in the complexity of these two tests can be clearly seen in the results. The same speedups are witnessed at a lesser scale on the GPU Long and CPU implementations.

Lastly, it was important to look at the speedup when compared to the number of paths run. When using the default PRNG, an increase for the first-hit time recordings, followed by a drop-off around 1E4 paths, this drop in production continued until at 2E5 the CPU would surpass the GPU. This was due to the increased memory cost as thousands of states were initiated for each kernel call. The 2-3x speedups seen by the default PRNG pale when compared to the speedups of 1200x, and greater found when using the Philox PRNG.

![Figure 4.6: Speed multiplier per number of paths run](image)

The data was obtained by running the program in a script that increased by 5000 paths per run. The decrease in the speed multiplier for the GPU was unexpected, so the program was re-run outside of the script at the exceptional areas. The time cost of those tests was consistently found to differ by about 10s from the previously recorded time when using the same test values. This
significant difference in time cost is also seen in the surrounding values when the script-run values are compared to singular runs. The original data was presented, as it is something to consider, but the loss in speed does appear to be an outlier likely caused by the setup of the script.

4.3 Robustness Comparison

The robustness of the program is a little harder to test. At some point, the program becomes unwieldy or possibly unusable due to time considerations. This section will discuss the points at which the speed became prohibitive for both the CPU and GPU, as well as any hard limits found. In many cases, the simulations can be run past these limits as time permits, but it provides an idea of what can be expected. The three determining factors when running these simulations are the size of the time steps, the length of the time run, and the number of paths run.

Table 4.2: CPU vs. GPU-L, 1E4 iterations, 1E-3 end time, first hit, no drift

<table>
<thead>
<tr>
<th>Δt</th>
<th>CPU-Drift</th>
<th>GPU-long</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E-5</td>
<td>0.16s</td>
<td>0.0006s</td>
</tr>
<tr>
<td>1E-6</td>
<td>1.62s</td>
<td>0.002s</td>
</tr>
<tr>
<td>1E-7</td>
<td>16.25s</td>
<td>0.013s</td>
</tr>
<tr>
<td>1E-8</td>
<td>161.6s</td>
<td>0.12s</td>
</tr>
<tr>
<td>1E-9</td>
<td>1599.3</td>
<td>1.03s</td>
</tr>
<tr>
<td>1E-10</td>
<td>N/A</td>
<td>10.0s</td>
</tr>
<tr>
<td>1E-11</td>
<td>N/A</td>
<td>100.3s</td>
</tr>
<tr>
<td>1E-12</td>
<td>N/A</td>
<td>1019.9s</td>
</tr>
</tbody>
</table>

The runtime appears to increase linearly with each exponential decrease in Δt. The simulations were stopped once the execution time reached over 1000s. The end time is inversely proportional to the time step used. For example, if a Δt of 1E-12 was used, but the runtime was decreased from 1E-3 to 1E-4, a runtime of about 100s seconds could be expected using the GPU-Long program. So, with a smaller runtime, a smaller step size can be used with little to no slow down, but if a long runtime with a tiny step size is needed delays will be found.

The limitations found on the number of paths that could be run started about 1E6 for the CPU as seen in Table 4.3, this was due to time cost not inability to run. For the GPU, when using the Wide version, at 5E6 paths it reached the 1000s threshold. The Long version easily runs the 5E6, in a speedy 5.12s. The device runs out of memory shortly after 5E8 on the GPU-Long implementation; it will not run at 5.5E8 paths due to memory constraints.
Table 4.3: CPU vs. GPU-L, 1E4 iterations, 1E-3 end time, first hit, no drift

<table>
<thead>
<tr>
<th>Number of paths</th>
<th>GPU-Wide</th>
<th>GPU-long</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0E6</td>
<td>1117.6s</td>
<td>5.08s</td>
</tr>
<tr>
<td>2.5E7</td>
<td>N/A</td>
<td>25.4s</td>
</tr>
<tr>
<td>1.0E8</td>
<td>N/A</td>
<td>102.5s</td>
</tr>
<tr>
<td>2.5E8</td>
<td>N/A</td>
<td>258.5s</td>
</tr>
<tr>
<td>5.0E8</td>
<td>N/A</td>
<td>530.0s</td>
</tr>
</tbody>
</table>

4.4 Reflecting Walls

Lastly, reflecting walls were added to the simulation. For this simulation, a receiver will be placed a given distance, $d$, away from the transmitter and the receiver is centered within the blood vessel, though it does not have to be. These walls represent the walls of the blood vessel and act as a reflecting surface for the molecules released. The blood vessel is assumed to be a perfect cylinder with a radius, $r$, the reflecting surface is also assumed to be perfect. Reflections were computed in real-time, time step by time step. If the next movement of a particle was found to be outside of the bounds of the blood vessel, the collision point can be calculated. Since the blood vessel is assumed to be a cylinder of radius, $r$, the start and end points of the movement in question can be massaged into a quadratic equation. It can subsequently be solved in order to identify the point at which the particle reaches the vessel walls. Using that collision point the reflection can be calculated and added to the vector ending at the collision point. This condition is reached using a radius based $if$ statement, that can be quickly skipped when irrelevant. The calculation for the collision point does take a non-trivial amount of time. With the implementation on the Wide set up, the execution time about doubled. Another route that could be considered for future work is the application of a Binary-search algorithm that will basically compare points on the vector and the radius until a match accurate to a given precision is found.

It can be seen that the mapping of the GPU results in Figure 4.7 does not match the analytical results, this is due to the addition of the wall to the no-drift algorithm. The analytic solution graphed, is the same equation that was used before for validation in the $\mathbb{R}^3$ open space results. However, with the addition of the vessel wall, the molecules are no longer in free space and therefore do not behave as such. What is instead seen is a reduction in the number of early hits with an increase in the number of hits later on. This is likely due to the confinements of the molecules to a space containing the
receiver. It would make sense the hit rate would increase with time as the molecules are herded in the direction of the receiver instead of having free range of the $\mathbb{R}^3$ space.

5 Discussion

A wide variety of approaches were used in creating this data. While some had significant effects on the time or robustness, other changes were barely distinguishable. The original design used was a wide approach. It was found to have a speedup of 2-3x from the GPU, but as the data increased the speedups decreased, to the point that it would take longer on the GPU that it would on the CPU. To provide a baseline for comparison, the calculation of $1E5$ paths recording only the first time hit would take 120s. Compare this with the 0.12s time that is accomplished now, and the time difference becomes apparent. The first instinct was the reduction of off GPU work. The wide algorithm as originally implemented, brought all paths back to the CPU where they were then either, printed or passed into logic loops to limit the amount of data output. However, by removing this code for speed test purposes, it was found that the increase in speed hit a limit at 104s. While this was a reasonable improvement, it was not realistic to assume that all of the CUDA copy and logic costs could be cut out. Even assuming they could be the difference between the CPU at approximately 160s, the GPU
at the new time of 104s, was a minimal 1.5x speedup.

With that consideration, the kernel was the next portion that was investigated. The thread cost was found to be relatively high. A reduction in the number of threads used resulted in much faster speeds. However, it did not allow the program to function as intended. While each thread was doing something, they were not doing enough to be of high value. It was more beneficial to break the calculations up among the hardware than it was to run sequentially, but the cost of starting a thread for a few calculations was high.

The other item in the kernel that had more than a basic math implementation was the PRNG. The default option from cuRAND was used. While the accuracy of the function had been reviewed, the time cost had not up to this point. It was found that if the PRNG set up was instead replaced with general float numbers, that the time cost of the entire program for the 1E5 path baseline reduced to approximately 15s. This was a substantial cost savings, but not necessarily realistic as the algorithms used were highly reliant on the randomness of those numbers. A few quick tests narrowed the excessive time cost to the state initialization. With that information, the state initialization was removed from every kernel call and instead merely run at each time step. While this did result in the speedup desired, it was found that the data produced had a high degree of correlation. Upon further review, it was found that while the seed was changing each time step, each thread received the same string of random numbers. Due to the design of GPUs, the threads did not read from that string sequentially, so a degree of randomness did exist, but it was definitely not enough for the design needed. The next route that was looked at was the implementation of a bit-shifting PRNG and other similar simple methods. The issue that continued to come up was either too much correlation or too much time. As the research continued, it became evident that many of the PRNG models were not designed with the GPU in mind or were inherently parallel. With this explanation, it also became apparent that a solution that used the threads more heavily may be a way to speed up the results. During this process, the Philox PRNG was discovered. It avoided the heavy initialization cost that many of the other algorithms used needed. The results were immediately visible, the simulation time dropped from 120s to 22s, a massive time savings and a speedup of more than 5x. This same algorithm was implemented in the long implementation and the simulation time dropped to sub-second, a speedup greater than 990x. Both setups passed the validation tests.

The compelling comparison with both routes validated comes with again removing as much of
Table 5.1: First-hit GPU-W vs. GPU-L Speed Comparisons for $R^3$ Receiver Test

<table>
<thead>
<tr>
<th>Paths Simulated</th>
<th>GPU-Wide-Base</th>
<th>GPU-Wide-noCPU</th>
<th>GPU-Wide-Memtrans</th>
<th>GPU-Long-Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1E5</td>
<td>22.3s</td>
<td>0.128s</td>
<td>2.17s</td>
<td>0.121s</td>
</tr>
</tbody>
</table>

the CPU work as possible on the Wide implementation. The Wide implementation is called by a loop on the CPU, so that part cannot be removed, but all logic and memory copy can be. The base implementation takes 22.3s to run, if everything CPU based is removed this time drops to 0.128s a considerable time improvement. Although, the removed parts would have to be implemented elsewhere in order for it to remain functional. The memory transfer also must occur, but if the logic is performed on the GPU, less data can be passed back. What is most fascinating is the minimal difference in speed between the particular execution styles. While the Wide project is a fraction of a second slower in the best case scenario, with an optimized implementation, it is not beyond thought that it to could reach sub-second speeds. The Wide implementation has been left as is for comparison purposes as well as to provide a method to print out entire paths. Memory does become a hurdle, if all paths are kept on the GPU; however, it is surmountable and may be something to look at in future work if this functionality is important.

The GPU used in this study was a single-precision GPU. Double precision GPUs do exist, but for this project, they were deemed unnecessary. A single precision GPU will store 32 total bits, one of those bits is a sign bit, and the next 8 are so-called exponent bits. This means that a significand precision of 24 bits can be expected. This equates to a maximum of $3.4 \times 10^{38}$ and a minimum of $1.2 \times 10^{-38}$. These ranges encompass all calculations made and expected to be made.

6 Future Work

There are many ways that this project can grow in the future. It can be customized to specific needs or tuned for increased speed as new technologies emerge. A few of the main things that may be looked at in future studies are the implementation of RBCs, the possibility of molecular reflection at the receiver and particle on particle reflection. Red blood cells are very influential on the paths of the molecule as discussed throughout this paper. It would add accuracy to the simulation if the RBC could be added. It would also provide a new way of understanding the expected routes of molecules in the bloodstream. There are several challenges in doing so; the RBC can change in shape, they also
continually rotate, and the collisions will need to be handled for the molecules as well as the RBCs. The receivers simulated in this paper assume 100% absorption, but in reality, that is unlikely. The receivers will be able to detect some of the particles, but depending on the design of the receiver, some of the particles may reflect or fail to be detected. Another possible future research opportunity would be particle on particle reflection. This would likely require multiple release points and a full data structure. The Wide algorithm could be updated to work nearly entirely inside of the kernel, or partially depending on output needs. Lastly, a multi-core version of the CPU code could be coded to work with a new PRNG and research could be done to find an intersection algorithm that may be better suited for the GPU, if one exists.

7 Conclusion
This thesis was able to realize extensive speed increases in the simulation of molecules for the further study of the molecular communication. It was able to increase the speed in many cases by 100 fold or more. It also provided the ability to calculate a number of paths, or simulation times previous viewed as highly time intensive, if achievable. Lastly, it proved the validity of the results and opened the door for further research, building on this platform. There are many routes that may be pursued in the future to continue this study.
Vita

Author: Tobias Cain
Place of Birth: Nampa, Idaho
Undergraduate Schools Attended: Spokane Community College

Eastern Washington University

Degrees Awarded: Associate of Arts, 2011, Spokane Community College

Bachelor of Science in Electrical Engineering, 2015,

Eastern Washington University
References


