EVOLUTIONARY SIMULATION OF LIFE USING CUDA
Technical Report #TR-20140505-1

Adam Call
Department of Computer and Information Sciences
Indiana University South Bend

CSCI-Y 790: Graduate Independent Study
Faculty Advisor: Dr. Dana Vrajitoru

1. INTRODUCTION

The idea behind this project was to create a simulation of the evolution of life in CUDA. In this simulation the creatures are individual entities that can interact with the world. Each has its own set of state information and DNA representing it. Through this DNA the creatures evolve via division and mating. The evolution of the DNA during reproduction utilizes crossover and mutation but does not have any sort of fitness function consideration. Each creature itself chooses when and how to reproduce, thus the creatures that live longer and/or reproduce faster will tend to survive better or be more fit. The creatures control their actions and interactions with the world via a recurrent neural network. The structure and weights of this network are encoded into the DNA as well as everything else about the creature.

The structure of this report is as follows. It starts with an overview of CUDA and the different facets of it that need to be considered when programing in CUDA. This is followed by a detailed description of the evolutionary model used in the simulation. It continues into the general structure of the data used to represent the simulation. The implementation of the simulation follows and covers both the high and low levels of the code. This section also discusses some of the design considerations with respect to the specific pieces of code being described. Lastly, the results of running the simulation will be discussed and final conclusions will be made.

2. CUDA DESIGN CONSIDERATIONS

"CUDA™ is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)" [1]. CUDA has the potential to greatly boost the throughput of an algorithm but typically necessities a significant redesign of the algorithm to realize this improvement. The reason behind this is partially due to the difference between sequential and parallel algorithms but more so to the very architecture of the GPU that needs consideration. In this project, only four facets of the CUDA architecture needed to be considered: CPU-GPU Memory Transfer, Global-Block Memory Transfer, Memory Coalescing and Warp Divergence.

The structure of the GPU consists of the shared outer memory and several inner processing units that each has their own memory, as seen in Figure 1. Each of these processing units can process tasks completely independently of the other units. Internal to each unit, multiple threads can be processed in parallel. The only limitation in this case is that each thread has to do the same thing but can do it on different data. This is referred to as SIMD (Single Instruction Multiple Data). Typically, the processing units also operate in a SIMD manner but do not have to. With more recent versions of CUDA, the GPU can have up to two different threads running in the CPU at the same time. The second thread in this case is almost always used to transfer data to and from the GPU while the main thread is processing the active kernel.

2.1. CPU-GPU Memory Transfer

CPU-GPU Memory Transfer refers to the time needed to transfer data between the GPU and CPU. With the exception of disk access, this is likely to be the lowest bandwidth pipe in the application. The issue is that the data must be sent over the PCI-E slot from the CPU to the GPU. This transfer takes a significant amount of time relative to the data transfers within the CPU or GPU. In response to this, data transfers between the CPU and GPU should be minimized. This project was designed from the beginning to keep as much of the simulation as possible internal to the GPU and to pay the majority of the transfer costs upfront during the simulation's initialization.

2.2. CUDA Memory Structure

The internal structure of the CUDA GPU is divided into many distinct parts. Only a logical representation will be presented here for the sake of understanding design decisions, as opposed to a more detailed physical description. This logical representation is illustrated in Figure 1. In the Outer Layer of the GPU memory, the layer accessible to the CPU, there are three memory Blocks: Global, Constant and Texture. Global Memory is the primary location to store data in the CPU. For this project nearly all simulation data is stored in Global Memory. In this report, Global Memory will also be referred to as Global. Global Memory is by far the largest Block of memory in the GPU, 2GB in this project's hardware, and is

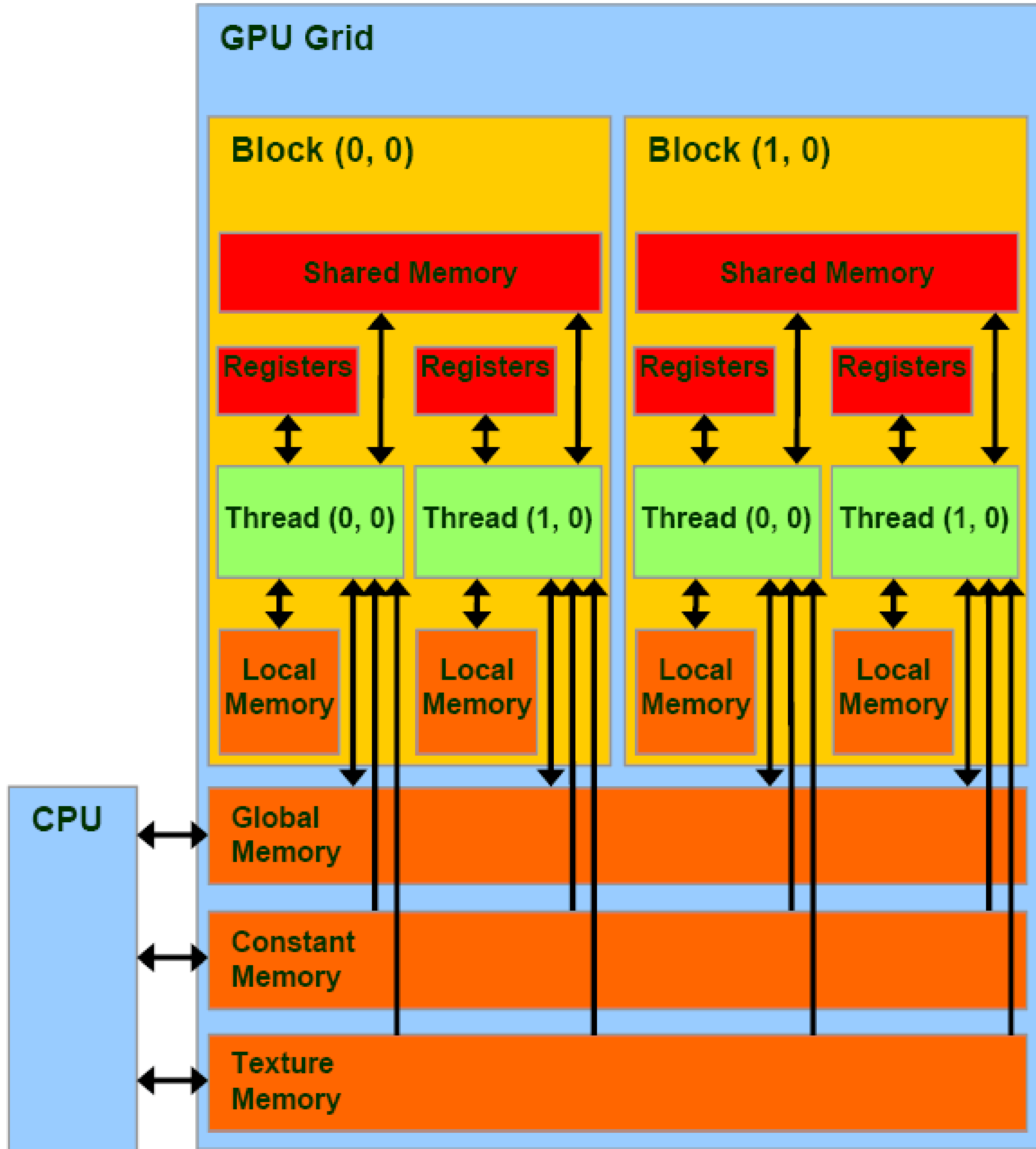

Figure 1: CUDA Memory Model [2]

writable from both the CPU and GPU. The price of having all this space is the limitation of only allowing operations to be performed at specific indexes. For this project the index period was 512 bytes. This limitation combined with the relatively slow transfer rate between Outer and Inner memory, is ultimately the source of the Global-Block Memory Transfer design consideration that will be discussed later.

Constant Memory is much smaller than Global, 64KB in this project's hardware, and can only be written from the CPU. Its advantage is that it can be efficiently accessed at a per word level, 4 bytes, as opposed to Global's specific indexes. The other benefit is that within a Block, to be described later, only the first access of a constant value requires an Outer Memory operation. Afterwards the Constant variable will be accessible at register speeds. One limitation of Constant Memory is that when defined, it only has a scope of the source file in which it was defined. Since this application has numerous CUDA methods (kernels) declared in individual files, many of the constant variables had to be defined multiple times for each kernel that needed them. This is ultimately a limitation of CUDA and simply had to be worked around. This was not a problem as there

was more than enough room in Constant for all needed copies of the constant variables.

Texture Memory is the last of the Outer Memory that is externally visible but was not used in this project and will thus not be described. The Inner structure of the GPU is first divided into Blocks. A Block represents both a logical division of the GPU's Inter Memory and of its processing power. When a kernel is executed, a copy of the kernel is given to each Block to run. Each Block has a pool of Threads that it uses to process the kernel. The CUDA API makes available several values to distinguish one Block and Thread from another from within the Block or Thread itself. Each Block has its own Shared Memory that is only accessible to the Threads within that Block. In older versions of CUDA compatible GPUs this was 16KB, for this project, 48KB ware available. The benefit of Shared Memory is that it provides much faster access times than Global, though not quite register speeds. The design consequence of this is that some sets of data could be used more efficiently if first transferred in their entirety to Shared Memory. This was typically done for any values that would be accessed multiple times within the Block to avoid retrieving the same value from Global many times. The other benefits of this method are described later.

The last two sections of Memory are the Registers and Local Memory. These two sets of memory are only accessible to specific Threads and are used for any of the locally scoped variables in the kernels. The difference between them is that the Registers are located close to the ALUs (Arithmetic Logic Unit) and are thus the fastest to access, while the Local Memory is part of the Outer Memory and slow to access. The reason for this is that the Local Memory is merely a backup for the registers. When a kernel has used all its registers, the Local Memory will instead be used to define any addition registers as needed. Care should be taken to avoid this, since it effectively converts the fastest memory access in the GPU into the slowest. In this report they will also be collectively referred to as Local since one is merely a backup of the other.

2.3. Global-Block Memory Transfer

As was mentioned previously, the access rates of the Outer Memory within the GPU are relatively slow compared to Shared and Local (Registers). In addition, Global can only be accessed at specific indexes. These aspects combined lead to the Global-Block Memory Transfer facet of CUDA that needs to be considered. Operations on Global memory can only happen on chunks of memory, 512B. This means any read or write operation must begin and end on a chunk boundary. While Global operations are possible off these indexes, additional operations have to be done in the background to fulfill them. What happens is that the entire chunk where the operation begins or ends is transferred in addition to the actual data requested. In both directions, the cache is used to convert the full chunks to and from the actual data of the operation. This leads to both wasted transfers and added processing time in the cache.

The remedy to this is to use pitching. When an array of data is declared, it always stars at the beginning of a chunk. When dealing with larger arrays of data, it is common to divide them up between the Blocks, for example by the rows of an array. The problem is that the data given to each Block most likely does not begin on a chunk boundary, which leads to longer access times. What pitching does is pad the end of each section of data with extra memory to fully fill the chunk. This allows each section to begin at a boundary and improves access time. In this project this method is used mainly when arrays of data are needed by the kernel. It is not used when single values are needed by the kernel.

The last aspect to consider is the situation when Threads need to access data in a non-deterministic way. Meaning, when the data needed is not known until the kernel actually runs and can change from run to run. In this project specifically, this mainly happens when the data itself determines what additional data is needed. This leads into the last benefit of transferring arrays of data from Global to Shared. Pre-reading the data needed into Shared allows the Global operations to begin on a chunk boundary and be carried out in the least number of operations, as opposed to accessing each value individually at some unknown location in the chunk. This method is also used in the reverse case where Shared is used to collect all the results from the Block before copying them all back to Global together.

2.4. Memory Coalescing

There are two sides to the task of transferring data between Global and the Blocks. Global's side of the equation was just presented but Block's side has its own considerations. These considerations have more to deal with how the Blocks process their Threads. In a Block the Threads are processed in Warps of 32 continuous sequential Threads. Logically speaking, the Threads of a Warp are executed in parallel. While a kernel is running, the Warps of the Blocks are processed as they become ready for processing. The implication of this is that 32 words can be effectively transferred at the same time. The GPU will actually try and merge Global operations within a Warp to utilize this ability. This merging of operations is known as Memory Coalescing. Without going into too much detail, the GPU will watch for continuous sequentially numbered Threads within a Warp requesting operations on continuous sequential locations in Global. When it sees this, it will combine the operations into as few Global operations as possible and feed each value to the appropriate Thread via the cache. This behavior works the exact some way in reverse and is typically used to copy results that have been gathering in Shared back to Global. The effect on design is mainly in how arrays are transferred between Global and

Shared. The technique is to use a loop that iterates by the Thread width (total Threads in a Block). Each Thread transfers the value in the array equivalent to its Thread ID up to the number of values to be transferred. This way the transfer between Global and Shared can be coalesced.

2.5. Warp Divergence

The last aspect of the CUDA architecture to consider is Warp Divergence. As was mentioned, Warps are groups of 32 continuous sequentially numbered Threads that run in parallel. What was not mentioned is that the Warp can only be doing one operation at a time. In other words, every Thread in a Warp has to be doing the same operation at the same time, such as an addition or multiplication. On the other hand, the data being operated on does not need to be the same and is typically not the same outside of flow control. This system works great until you reach a flow control statement (i.e. if, for, while, etc.) The problem with flow control statements is they can send the individual Threads down different execution paths. The way this is handled is the individual paths are processed in sequence until the end of the section that divided the flow, for example the '}' in C++. At this point the Threads are brought back in order and continue to run in parallel. The branching of these Threads is called Warp Divergence.

The design consequence of Warp Divergence is mostly to be aware of it and limit the branching of the code. If it is unavoidable, then it is best to keep the divergent sections of code short. One thing to clarify is the difference between a diverged Thread and an inactive Thread. An inactive Thread is a Thread that either has no work and is waiting for the other Threads to catch up to it or a Thread that has finished processing of the kernel. The most common situation when this happens is when a Thread is not needed for the current operation. In this case the Thread just sits inactive until it is needed again. An inactive Thread does not harm the performance of the other Threads and is a normal part of kernel execution. In some cases the problem might lend itself to a Warp level division of labor. In this case the divergent flows of the kernel each get their own Warp on which to process. This way the individual Warps do not diverge while the flows between the Warps do. This technique is slightly more difficult to implement and in this project at least, was only practical for one of the data structures.

3. EVOLUTIONARY MODEL

The creatures in this simulation can best be described as ants in the sense of the ant colony model [3]. Each creature has a recurrent neural network that controls its individual behavior each cycle. These creatures can interact with the environment but cannot interact with each other beyond breeding. Further creature to creature interaction was delegated to future work. This simulation follows an evolutionary algorithm where the population of the creatures is not of fixed size and can grow and shrink as creatures die and reproduce [4] [5]. The creatures themselves are represented as DNA sequences. These sequences encode all the parameters defining the creature. With respect to the recurrent neural network, both the weights and the structure of the network are encoded [6] [7]. The creatures in this model actively choose when to breed and can choose between division or duplication and mating when they do. When mating, a bounded two point crossover is used on the DNA segments [8] [9]. In addition to crossover, Hill-climbing is used in the form of mutation for both operations. When performing these operations, the DNA segments are considered to be the smallest divisible units of the DNA sequence. These indivisible DNA segments are also known as genes. This means that no operation to the DNA is allowed to split a DNA segment or produce a partial segment.

The method of parallelization for this simulation is multi leveled to match the structure of CUDA. CUDA can divide the problem into at least three levels of parallelization: Block, Warp and Thread. For most of the operations in the simulation only two layers of parallelization can be achieved. For operations relating directly to the creatures, such as deciding what to do, the Block level of division is the individual creatures and the Thread level is the individual calculations. Warp level division is only used when the operation can be divided three ways, which rarely happens. The below sections will go into more detail about the world and the creatures that live in it.

3.1. World Model

The world is represented as a space containing varying quantities of energy. This space is defined via three primary constants. These three values are effectively natural constants of this world. The first is the dimensionality of the world. The world can have as few as two spatial dimensions and as many as the hardware can support. The next constants are the number of energy types and energy frequencies which describe the different types and frequencies of energy available. The energy contained in the world has both of type and frequency and is referred to as Radiant energy. This energy symbolizes various resources available in the world that the creatures depend on for survival.

The space in the world is defined both in a discrete and a continuous manner. To clarify, discrete uses zero to max integers to represent specific places in the world while continuous uses zero to max reals to represent specific places in the world. Places in the world are referred to as Locations when using discrete numbers and Positions when using continuous numbers. There are fundamentally two types of interaction with the dimensions of the world, energy and physics. Physics deals with the movement of the creatures and uses the continuous representation of the world. Energy on the other hand must use the discrete

representation since it is infeasible to model the energy map in a continuous manner. Every Location in the energy map thus has a distinct quantity of energy at that specific Location. Other parts of the model dealing with energy continue to use the discrete representation.

Energy type is used to represent the difference between say light, sound and chemical energies. Frequency on the other hand is analogous to the different frequencies of light or sound. The energy in the energy map is considered transient energy or Radiant energy. Radiant energy is the primary thing the creatures interact with in the world. Creatures can actively choose to sense different type-frequencies of energy. In this manner energy types represent all the senses the creature has, for example sight, smelling, hearing, etc. The creatures can also actively choose to absorb and store different energy type-frequencies. Within the creatures, energy does not have a frequency associated with it. Frequency is only used when dealing with the radiant energy. This stored energy is what the creature uses to perform all its actions and maintain its neural network.

One of the problems identified early in the design stage was the issue of how to persist the creature's impact on the world since radiant energy is naturally transitory. In other words, it would reset to zero between cycles. The original plan was to implement a ground concept to store energy in the world. Due to time constraints this was not implemented but a simplified method was used instead. The alternate method was to use a percent rollover of the radiant energy between cycles to persist the effects of the creatures. The effects of the creatures are primarily the absorption of energy from the world. This adds a disadvantage to absorbing energy in an inefficient manner as the creatures could starve from wasting energy.

3.2. Creature Model

Contained within the world is a population of Creatures. The size of this population is not fixed and can change overtime as different factors affect birth and death rates. Due to hardware limitations this population is capped at some hard limit. The ideal case for this population is to have it stay relatively stable at some sub max value.

The creatures are primarily a recurrent neural network. This recurrent neural network controls all behaviors of the creature via actions that can be defined on each node. These actions include things like moving and feeding but also extend to sensing the environment. Via the sensing actions the creature can see the energy levels of specific type-frequencies in a region. Most actions also apply to specific regions that are part of the creature definition.

Regions are specific to a creature and are defined relative to the creature's location. They use a signed discrete representation of the dimensions of the world. Since the regions move with the creature, they can be partially or completely outside the valid dimensions of the world. When this happens the invalid sections of the region are ignored the region is truncated to fit in the valid world dimensions.

The fundamental parameters that describe a creature's physical behavior are stored energy and momentum. Stored energy is the resource that creatures use to preform actions and maintain their neural network. It is also used to calculate the creature's mass and radius. Momentum is fundamentally mass times velocity and is stored instead of velocity because it is not dependent on mass. In physics momentum is preserved, meaning the total momentum of a system will stay the same in the absence of an outside force. This way the mass (stored energy) of the creature can be changed without having to recalculate the velocity every time.

The motion of creatures is fundamentally determined by dividing their momentum by their mass to derive their change in position. Changes in their motion are thus directly related to their momentum and mass. Changes in mass only have the effect of amplifying or dampening motion so will not be discussed in detail.

As for changing momentum, there are five different factors that will affect it. Firstly the creature can choose to move by converting stored energy into kinetic energy and in turn into momentum. This choice is an action and is activated by the neural network. The second and third ways are via collisions with other creatures and with the boundaries of the world. Collisions in this system are modeled as simple directional springs to improve stability and allow the creatures to overlap. Additionally, collisions damage the creatures participating in them in a manner that is relative to the impulse applied. Damage is a reduction in stored energy. The last two factors are drag and friction which act to dampen motion.

All creatures in this model also have a DNA representation. This DNA sequence consists of segments containing four values each that map to both specific attributes of specific nodes or regions and to the general information about the creature. All facets of the creature can be encoded into the DNA. Thus both the weights and the architecture of the neural network used to control the creature are represented in the DNA. For the most part, segment order in this sequence does not matter since each segment includes the ID of the component it belongs to. Where it does matter is when you have two segments setting the same value. In this case the segment later in the DNA sequence overwrites the earlier ones. Lastly, the DNA sequence need not be complete; anything not explicitly specified in the DNA is set to default values. When creatures are initialized from an un-encoded form, DNA is generated for them containing any values that are not equal to the default values.

Creature death happens when the creature does not have enough of a specific energy type to support its maintenance costs at the end of a cycle. Dead creatures are removed from the world and their location in the data structures is

marked as empty, meaning it can be reused by a new creature. The creatures can also die by actively choosing to die. There can be benefits to a species not being long lived such as rapid evolution.

In the same manner reproduction happens via actions. The creatures actively choose to reproduce via division or mating. The difference is that mating uses both mutation and crossover while division only uses mutation. Crossover happens on two crossover points. These crossover points are percentages of the total DNA segment length of the parent creature. This allows for DNA segments of differing lengths to be crossed. Additionally constraints are placed on these points to ensure that the total crossover percentage is within a settable range. These percentages also round to the nearest segment border. Partial segments have no meaning in the model so are not allowed.

Mutation is performed for division and after crossover for mating. It is performed a set number of times and has four types: move, copy, delete, change. For each of these types a segment is randomly selected from the DNA and for move and copy a random location is also chosen. For move, the chosen segment is moved from its current location to the target location selected. Since a segment later in the DNA sequence is less likely to be overwritten, move has the effect of altering how likely the segment's value is to be used. Additionally, segments that are located closer to each other in the DNA sequence are more likely to stay together during the crossover operation and vice versa. This has the potential to separate desirable and undesirable segments more easily during the crossover operation. Copy does something similar but only moves a copy of the selected node. This leaves the original segment where it was and increases the DNA's length. Since copy effectively moves the segment, it has the same effects to the creature as move does. Additionally, copy increases the probability that a segment will be passed on in a crossover by increasing the number of copies of the segment present in the DNA. Delete simply removes the selected node, decreasing the DNA's length and causing the default values to be used in place of the deleted segment. Change randomly selects an attribute of the segments to exclusively or with a random value. Exclusive or is chosen since it has a balanced truth table. This means that, given random bits, it will product '0' half of the time and '1' the other half.

As a final note, this model has no explicit fitness function. The likelihood of a creature's DNA being passed on is directly dependent to how long they live and how often they breed. Similarly, there is no concept of generations as the creatures choose when they want to breed. Lastly, mates are chosen randomly with no preference given to proximity. This was done for simplicity sake and could be improved in future work.

4. Simulation Data

The simulation data is largely divided into three sets of information, the creatures, the world and the running conditions. The configurable portions of this data are located in an XML configuration file that is loaded by the application. At the top level are the running conditions of the simulation. These consist of the CUDA device to use, the number of cycles to run, the maximum number of cycles to wait between logging points and whether to write any new files generated with only the creature's DNA or also in a verbose readable form. The CUDA device refers to which GPU in the system to use if there is more than one. Number of cycles is how many cycles to run in total for the simulation. The max period for logging is also referred to as cycle batch. The system that runs the cycles will attempt to run this number of cycles but will stop prematurely if an actionable event takes place in the simulation. The last option allows a more readable form of the creatures to be written when requesting the simulation to write the ending configuration file. Note: if both the DNA representation and the verbose representation of a creature are present, the DNA one will take precedence.

4.1. World Data

Through the world, the global constants of the simulation are setup. For the dimensional constants, the world takes the number of dimensions as a parameter as well as the width of each dimension. In the simulation, the dimensions go from zero to the given maximum and are represented as either a 32bit integer or a 32bit floating point number. To reiterate, the coordinates of something relative to the spatial dimensions is its Position and is stored as a float, while the coordinates of something relative to the energy maps is its Location and is stored as an integer.

The other major property of the simulation is energy and its varieties. Energy can be defined in the world in a number of types of which each can exist in a number of frequencies. When referring to energy with both a type and a frequency, it is described as an energy type-frequency as opposed to energy with only a type, which is an energy type. A reference to only energy typically refers to energy without a type. Creatures store some amount of each energy type and consume energy each turn both through their maintenance costs and their actions. The amount of consumed energy is without type. The DNA of the creature determines what percentage of the total is paid with each energy type.

The world also contains the radiant energy maps or simply energy maps. These maps of energy are what the creatures detect when they try to sense the environment around them. Creatures detect different statistics about specific energy type-frequencies in regions of these maps. The implication of this is that the creature could specialize its sensors to only detect specific energy types or

frequencies. Creatures can also absorb these energy type-frequencies from the world around them.

There are actually two radiant energy maps, the main one containing the actual radiant energy that the creatures directly interact with and a secondary one to hold the percentage changes that need to be applied to the main map during the next update due to creatures absorbing energy. The reasoning for the second map is explained later in the "ApplyRegionEnergyDelta" kernel description.

The energy maps have a different structure to their data then would be expected. Fundamentally they have every spatial dimension in them plus a full set of energy type-frequencies at each spatial position. Energy frequency is the innermost dimension while the first spatial dimension is considered the outermost dimension. Within the array representing the multi-dimensional structure, the innermost dimension would be the one that only requires an offset of one word to increment its index. The outermost dimensions would require an offset of the product of all lower dimensions to increment its index. The complication in this structure of data is that it is flattened for the sake of pitching. In the simulation every spatial dimension but the lowest is folded into the upper dimensions and are collectively called flattened rows. Each row consists of a single index of the second innermost spatial dimension plus all the energy type-frequencies for that row. In other words each row has a width of the innermost spatial dimension's width multiplied by the number of energy types and energy frequencies. The structure is then pitched on these rows to align them to the Global memory's specific indexes.

There are two additional settings related to these energy maps and those are the ambient energy and the percent rollover. The percent rollover is the amount of radiant energy of each energy type-frequency that is carried over from the previous cycle into the new cycle. This allows for the changes the creatures make to the environment to have some lasting effect. Ambient energy, on the other hand, is the amount of each energy type-frequency to add to the system each cycle. It is effectively the sun shining down on the earth. As long as the percent rollover is between but not equal to zero or one, these will eventually reach an equilibrium point given no creature interaction.

The last values defined within the world are the system limits. These are the maximum number of creatures, the maximum number of nodes per creature, the maximum number of regions per creature and the maximum side length of each region. These are internally validated against the hardware to see if the simulation is runnable.

4.2. Creature Data

The majority of the data in the simulation deals in one way or another with the creatures. Each creature has three base pieces of state information about it. The first is its Position in the environment and intern calculated from this its Location in the energy maps. The second piece of information is the creature's stored energy types. Again this value is used to derive other values, specifically the creatures mass and radius. The last is the creature's status. This is mainly used for signaling on a creature wide level whether the creature is dying, reproducing or uninitialized. With respect to uninitiated creatures, there are several values that mark a creature as uninitialized: a null status, a zero radius, having zero nodes and having zero regions. The last two can happen naturally but the main purpose of these marks is to skip work that does not need to be done. A creature evolving with no nodes or regions or a creature being uninitialized does not matter as long as it is recognized that there are no items to process. The creatures have one additional configurable property at the creature level and that is the percentage of each energy type to use for paying the maintenance cost. The maintenance cost is based on the number of nodes and links in a creature and is expressed in only energy without type. These percentages divide up the total cost among the different types. This allows the creature to potentially evolve out of the need for a specific energy type and actually live on even when it has negative of that energy type.

Each creature has a set of interconnected nodes that form a recurrent neural network to control the behavior of the creature. These nodes each have an action associated with them. The specifics of these actions are described later in the "ResolveActions" kernel. Most of the actions preform their task on a region of the radiant energy map who's Id is included in the node definition. The actions also have dimensional parameters which are typically a direction in which the action happens. Note: this is only a direction and not a location (region) for the action to happen. In addition, the action has a set of energy type-frequency parameters. The value these parameters encode varies from action to action but is always something dealing with energy.

Each node also has a charge associated with it that is the sum of the incoming signals from linked nodes. This charge is used with the activation function to determine if the node is active. There are actually several different activation functions with the choice and configurations of the used method dependent on a pair of parameters call On Charges. Like with the maintenance energy at the creature level, each node also has a set of percentages that controls how much of the activation cost is paid by each energy type. The activation energy cost is dependent on a fixed amount for the node and the number of links leaving it. The definitions of the incoming links for each node consist of the node the link is coming from and the scale and offset to apply to the charge of the linked node. This incoming scaled and offset node charge is added to the target nodes charge. The nodes only define their incoming links and not their outgoing ones.

The last items defined for the creature are the regions. Each creature can have some maximum number of regions defined for it that are located relative to the creature's Location. The regions are defined as a minimum and maximum offset in each dimension relative to the creature.

Since the creatures can move around, the actual location of these regions in the energy maps can change, allowing for the creature to interact with different portions of the world. One thing to note about regions is that they can exist outside of the current world and could thus be partially or completely invalid. When the regions are processed, the portions of the regions outside of the world are ignored from consideration.

5. SIMULATION IMPLEMENTATION

The following sections will go into the details of how the different pieces of the program operate. In addition, design considerations and reasons for specific choices will be mentioned for the specific pieces of code being talked about.

5.1. EvoSim

EvoSim is the externally visible layer of the simulation. It handles all the high level interactions with the simulation. Through it a simulation can be loaded, run and logged. In this project the interactions with EvoSim are handled by a simple main function. This function starts by instantiating an instance of EvoSim. This is a parameter-less constructor as the configuration of the simulation is stored as XML data. Loading this XML from a file is the next step of the simulation, followed by the validation of the loaded file. The creatures in the simulation can be stored in either their verbose form or as a DNA sequence. When both are present the DNA sequence takes priority. When validating the configuration, the GPU is checked for compatibility with the simulation and the configuration parameters are checked to validate that they will not overrun the limitation of the Shared memory. EvoSim's Malloc is called next but it will require a successful validation before running without returning an error. The last step to setup the simulation is to call MemcpyHostToDeviceAll to copy all the data about the simulation to the GPU.

Running the simulation is simply a matter of setting the values you would like to be logged and calling RunCycles. The entire configuration about the simulation is in the XML configuration file. The last function of note is SaveConfig that allows you to save the current configuration of the simulation at any point. This is mainly to allow you to analyze the living creatures at the end of the simulation or start a new simulation from the same point. The remainder of the main function is just cleanup operations.

There are a few more functions in EvoSim that warrant some additional explanation. The first are LoadTest and RunDebug. This pair of debugging functions were used before the XML loading was implemented. LoadTest loads a default configuration of the simulation and RunDebug runs the simulation with some extra print statements. RunDebug does not support logging of the simulation results.

Loading of the XML configuration files starts with LoadConfigFile. This function handles the general data not about the world or the creatures. The world and creature tags are handed off to their respective classes to be decoded directly into their respective member variables.

The job of RunCycles is three fold. The first purpose is, of course, to run the cycles of the simulation and the second to log the data that has been requested to be logged when SimCore returns from a cycle batch. The maximum size of a cycle batch can be specified in the XML and is the maximum number of cycles SimCore will run before stopping for logging. SimCore will stop prematurely if an actionable event happens before all cycles are run. The third purpose, happening in the last step, is to handle the actionable events.

Actionable events are reproduction and death. These are handled at this level due to their sequential and highly complex nature. Another problem is that the handling of these events can cause memory allocation and deallocation. The last issue is that the reproduction algorithm needs access to the DNA representation of the creature which only exists in the high level representation of the simulation. The first step of handling the actionable events is to update the variables in the high level representation from the low level representation. When this is done, creatures are checked for a status of death and deallocated if need be. The deallocation code handles marking the creatures and their regions and nodes as deallocated so that the kernels can skip them.

If the reproduction flag was set or this method was called due to a full batch of cycles being run, the reproduction code is processed. One of the problems encountered when actually running the simulation was sterility. Since the creatures can choose to reproduce they are able to evolve to a point of being serial. To counter this, forced reproduction is done with the original fertile creature to keep the population from going sterile. Following this is a pair of loops to check every node of every creature that has a status of reproduce.

There are two reproduction events, Divide and Mate. The difference is that Divide only mutates the original creature while Mate preforms a DNA crossover with a randomly selected creature or the base creature in the case of sterility. The properties of the actions that generated the event actually control some aspects of the reproduction. Firstly, the dimensional parameters of the node designate the relative position where the new creature should be placed. The energy parameters of the node designate the percentage of energy that should be given to the child. Some percentage of this transferred energy is also lost as a cost of reproduction. After this, the original creature is no longer needed so its updated energy can be written back to the low level representation of the simulation.

Evolution is handled two ways, mutation and crossover. Both Mate and Divide have mutation but only Mate does any crossover of DNA between creatures. Crossover is

handled first so that both methods can use the same mutation code. For any creature that mating applies to, crossover starts by randomly selecting a second parent from the other creatures. The crossover is handled by a pair of percentages. Two different random numbers are chosen between zero and one (inequality is enforced). These values are used to split the DNA of the two creatures at percentages of their lengths. This was done to remedy the problem of varying length DNA. Before splitting, these values are checked to make sure the amount of DNA crossover between the two creatures is within a set window, 0.25 to 0.75 for this project.

The DNA is represented as a linked list of segments. Each segment contains 4 values which are used to varying degrees. Parameter type is the first and is an enumeration of all the different configurable creature parameters. Next is the segment info which is used as an index or ID of the element within the creature, (node, region, etc.). Sub segment info is any further categorization of the parameter, for instance dimension, energy type, etc. The last is the actual value which can be read as an unsigned long, long or float. When reading DNA values, a modulus is preformed against the limits of that parameter to put it into a meaningful range.

The crossover of the DNA sequence is handled by three iterator loops. The first copies the first creature's DNA from the zeroth segment to the one designated by the lower of the two percentages. The next loop copies the segments from the second creature from the lower percentage to the higher percentage. The last loop finishes off by copying again from the first creature at the high percentage till the end. In addition, if the lower and upper percentages have been reversed, the creatures are also reversed when the percentages are swapped back into order. This algorithm allows for a wide variety of different crossovers to happen.

The second method of evolution is mutation. The number of mutations to perform is designated by a constant. For each mutation, a random node is selected and one of four mutations is performed on that node. The first is simply to randomly move the node elsewhere in the list. This does not change the node in any way and merely changes where it is located in the DNA. The next method is to copy the current node to some other location in the code. The next and simplest mutation is to just delete the node. The final mutation is to randomly choose one of the values in the segment to randomly change. This modification is done by performing an exclusive or on the value chosen with a random value. The exclusive or was chosen because it does not favor 0 or 1. Three out of four of these operations require randomly changing the positioning of segments in the DNA. A linked list was chosen to represent the DNA specifically because of these mutation operations. The penalty is that the individual segments must be indexed too. A utility function was created for this purpose and simply returns the iterator into a linked list some number of steps in.

With this, the DNA for the new creature has been created and can be decoded by the new creature. This handles fixing critical problems in the creature due to any problems in the DNA. Specifically, it keeps track of hanging links and region references. When it finds a hanging link it creates an object to hold the other end of the link even if the link is the only non-default piece of information about the object. It also handles the scaling of values read from the DNA into the proper ranges. After all creatures have been checked for reproduction, the data about the creatures is updated in the lower level code and control transfers back to the GPU.

5.2. SimCore

The purpose of SimCore is ultimately to handle all interaction with the GPU. Its first responsibility is that of a memory manager. The high level representation of the simulation contained within EvoSim would be very inefficient if used in the GPU. To improve the performance of the kernels, the data must be structured into such a way that it makes it more efficient to transfer between the different sections of memory in the GPU. When using Object Oriented methods, the data is structured into an Array of Structures form [10]. Structures in this phrase are meant in the general sense and not in the literal sense. What this means is that data is arranged into an array where each index of the array has many values. While this data structure is acceptable in a sequential system, it makes Memory Coalescence impossible. Only data that is stored sequentially can be coalesced for transfer. Also, each kernel will only need a subset of the data. This leaves one of two choices. Either transfer unneeded data or jump around in memory only taking the pieces of each structure that are needed.

The alternative method is Structure of Arrays. Again, structure is not meant in the literal sense. With this strategy, each creature represents an index or a set of indexes in an array. These individual arrays only contain data about one of the properties. Some other structure, SimCore in this case, contains these arrays. The benefit is that now the data for each parameter is continuous and can be coalesced for transfer within the GPU. Thus SimCore needs to contain a low level, Structure of Arrays, representation of the Simulation data to be transferred to the GPU.

Several utility classes have been created to handle allocation and deallocation of memory for both the GPU and CPU simultaneously. The simplest is DeviceArray, which is just a single array of some type. The class defines several Malloc choices. There are really two choices to make with respect to which Malloc function to use: first, does the array need to be pitched and second, does an initial value need to be set. Pitching an array is when the rows of a two dimension matrix are offset to correspond to the Global Memory indexes, typicality by 512 bytes. The advantage is improved transfer times of individual rows at the cost of

wasted memory. The class also defines a free method that will clean up the memory on both the CPU and GPU. Lastly, this class exposes methods to copy data between the GPU and CPU arrays.

The next class is DeviceJaggedArray that is simply an array of arrays. More specifically, it is an array of DeviceArrays. It defines all the same functions as DeviceArray with the addition of per item versions of each behavior. The last class is the DeviceMultiArray which is only really used for the radiant energy maps. What is special about it is that it allows for a multidimensional array to be defined and then pitched on any of its dimensions. Other than that, DeviceMultiArray defines much of the same functionality as the other two classes.

These three classes are used by SimCore to define all arrays of data contained within the GPU. The allocation and initialization of these arrays are handled by SimCore's Malloc and MallocCreature functions. Malloc handles the allocation and initialization of all non-creature specific arrays. This includes the arrays about the environment, the top level of the jagged arrays and the non-jagged creature arrays. The non-jagged creature arrays are of fixed size based on the values given to the simulation at construction so can be allocation without any relation to the individual creatures. The top levels of the jagged arrays are the same way, only depending on the maximum number of creatures.

MallocCreature handles the allocation and initialization of the individual creatures but also has to deal with some conversion of data during the initialization. The special cases in this method are the creature's nodes and regions. In the case of the nodes, each node must be allocated and initialized individually. While doing this, the maximum number of input links must also be found. This max links per node value is used as the height of the pitched links matrix. With the array allocated, an internal function is called to convert the given linked list form of the linked nodes into a compressed form. The specifics of this compressed form are described later.

The regions must also be converted to the lower level representations. The high level representation of the regions is a list of the left and right offsets relative to the owning creature, which defines the boundaries of the region in each dimension. Regions within the GPU only really deal with Location as opposed to Position. To more easily process a region in the GPU, the boundaries are converted into a list of relative offsets to rows that need to be checked in the radiant energy map. The first step is to find out how many rows are needed total so that the memory can be allocated. The second step is to loop through the dimensions, recording every relative offset row needed. The innermost dimensions range is stored separately since it is the same for all rows in a given region.

The last method of note in SimCore is RunCycles, which as the name implies, runs cycles of the simulation. Specifically it will attempt to run the number of cycles it is told to. The method will also terminate prematurely if an actionable event, (reproduction and death), happens before the total number of cycles is run. The method has three by-reference parameters to return the actual number of cycles run and which event was found. The kernels run in the following order: CalcRadiantEnergy, CalcRegionValue, CalcNodeCharge, CalcNodeActive, ResolveActions, ApplyRegionEnergyDelta, ResolveCollision, ResolveMovement and CheckReproductionAndDeath. CalcRadiantEnergy needs to be first so that the radiant energy map can be initialized before it is used in the first cycle of a simulation. CalcRegionValue follows this to calculate the new region values from the updated radiant energy. CalcNodeCharge and CalcNodeActive handle the processing of the creature's neural network. The task of calculating the node charge and checking for activation was divided to reduce shared memory usage. ResolveActions handles the processing of all the actions caused by activated nodes. ApplyRegionEnergyDelta's job is to apply the energy deltas from each region to the secondary radiant energy map of percentage changes. These changes in energy are due to the actions in the previous kernel. ResolveCollision handles the detection and resultant impulses of all collisions between creatures but does not actually apply them. ResolveMovement collects the impulses from collisions and actions and does the physics computations to resolve the creatures' new positions and momentums. In the end CheckReproductionAndDeath handles the deduction of the node and link maintenance energy from the creatures and the detection of reproduction and death events.

5.3. Kernels

In CUDA, a kernel is a function that is run by the blocks within the GPU. Fundamentally, a kernel is the "Single Instruction" in SIMD that the GPU runs in parallel. It is defined similarly to any function with parameters but no return value. These parameters are either values or pointers to locations in the GPU's memory. By design, nearly all of the actual calculations for the simulation happen in the kernels. This was done to maximize the opportunities for parallelization of work and minimize the overhead associated with moving data between the CPU and GPU. The only task that was deemed impractical to implement in the GPU was the handling of reproduction and death. The two main reasons were that reproduction required the high level representation of the creature and both reproduction and death caused memory allocation and deallocation.

5.3.1. General form of the kernel

With the more recent versions of CUDA, the support for kernel linking was added. This functionality adds ".cuh" header files in addition to the ".cu" source code files. This functionality allows for kernels to be divided up into smaller files as opposed to existing monolithically in the same file as would have to be done previously. The down side of this

is related to calling the kernels and the constant memory variables. A method was not found to directly call the kernels from outside of the CUDA files they are defined in. The solution was to simply add a C++ function in the same file to call the kernel. With respect to the constant memory variables, the CUDA documentation [11] defines that the scope for the "__constant__" tag used to define the constant memory variables, to be only the file in which the variable is defined. This unfortunately necessitated the redundant declaration of needed constant memory variables in every kernel ".cu" file that needed it.

Each kernel has six functions defined for it based of the root of the kernel name. The kernels use the root name with "Kernel" post fixed. The C++ wrapper uses this root name directly and has an additional three parameters beyond what the kernel requires. These parameters are the Grid size, Block size and size of the dynamic Shared memory Block needed by the kernel. The last three functions are all post fixed with "SharedMemoryNeeded" and prefixed with either "Calculate", "Constant", of "Total". "Calculate" returns the variable byte size of shared memory needed based on given parameters. "Constant" returns the non-variable component of the shared memory needed and "Total" simply returns the total of the two. The shared memory size passed into the wrapper function is actually only the variable component of shared memory as the kernel defines its own constant shared memory.

The kernel naming conventions follow a common pattern. They start with the declarations of all needed variables. The variables in a kernel have one of 3 prefixes or no prefix and use camel case. The parameters passed into the kernel have no prefix and are either pointers to Global memory or values. The other 3 prefixes are used to designate where the variable exists in GPU memory. They are 'g' for Global, 's' for shared and 'l' for local. The prefix consists of one of these letters followed by an underscore. The Global variables are pointers into Global memory. They are used to sub index the Global memory pointers that are passed in as parameters. The Shared variables are either pointers into the dynamic Shared memory Block or non-dynamic Shared values defined within the kernel itself. The Local variables represent the registers and the backup Local memory for when there are too many registers. They are used as inter-kernel processing variables. Following are examples of this naming convention for a parameter, global pointer, shared pointer and local variable in that order: nodeCharge, g_nodeCharge, s_nodeCharge, l_nodeCharge.

The reset of the kernel can be dived into to three broad categories: setup, processing and update. The main purpose of the setup sections is to initialize the Shared memory. This can mean both coping data from Global and setting default values. Values copied from Global to Shared are typical done via a multi-Threaded loop. Simply put, the sequential Threads each copy one value in order before wrapping around to copy more if need be. This improves memory transfer coalescence. Typically, local values are also initialized in this section but need not be. The reason for this is that Shared memory usually contains arrays of data while Local memory usually contains single values. A setup section frequently ends with a sync to make sure all the data has been transferred before continuing. A kernel can have several setup sections for each division layer of the work. The most common places are directly preceding the Block and Thread loops.

The processing sections vary a lot by the different tasks they perform but contain some common elements. Frequently a loop starts with a conditional check to see if the current item should be skipped. There are many cases where an item is invalid and should not be processed. Each kernel typically has several nested processing loops. These are typically based on the Blocks and Threads but can also divide the problem by Warps for some of the more complex item structures. Further description of each kernel can be found later in this report.

The last section is the update section. The purpose of the update sections if to write values back to Global. Frequently, Shared memory is used by the processing loops to store the results locally so that they could be copied back to Global in an efficient manner. This efficient manner is again a small Thread loop that assigns subsets of the values to each Thread. Just like the setup section, there can be several update sections. The update sections typically mirror the setup sections and appear directly after a processing loop. Frequently an update section begins with a sync to make sure all data is ready to be transferred.

5.3.2. CalcRadiantEnergy

As was stated, the environment the creatures live in is represented as a multidimensional energy map. The purpose for the CalcRadiantEnergy kernel is to update the values for the radiant energy map. In the current implementation there are three contributions to radiant energy. The first is a settable value of how much energy should roll over from cycle to cycle. The next is how much additional/ambient energy is being added to the system each cycle. The last is how much energy the creatures absorbed from the radiant energy.

Since this kernel deals with the radiant energy map, the work load for the Blocks is divided by the flattened rows of the energy map. Once again flattened rows are all dimensions above the first reduced to one very big dimension. The first thing done is to copy the ambient energy being added to the system from Global to Shared. From here the kernel splits into two cases. The difference is whether there is a percent rollover of radiant energy in the simulation. If there is not then several steps can be skipped. In this initial implementation only the case including percent rollover is used. The original idea was to have land in the simulation. Pieces of land would consume spaces and also store and release energy. Percent rollover was a simpler way to get the radiant energy persistence that land

would have given. This is important because it allows creatures to have a lasting effect on the environment. The other source is the energy the creatures have absorbed. An idea for future work is to have the creatures emit energy based on their current energy levels. The idea was they would have some control over the frequencies of energy but not over the quantity. Thus, a whole stealth aspect could be born from the combination of sensors and creature emissions in different frequencies.

For the percent rollover case, the first thing done is to copy the old radiant energy and the percent rollover values from Global to Shared. In the main Thread loop each Thread handles a subset of the total energy types and frequencies per row. In other words, the rows in the energy map have the number of energy types multiplied by the number of energy frequencies multiplied by the number of columns of floats in them. In this calculation the energy absorbed by the creatures is removed before the percent rollover is applied. Next, the incident ambient energy is added. This new value is then written to Shared. In the non-percent rollover case, the new radiant energy is directly equal to the incident ambient energy. Lastly, the new radiant energies are copied back to Global in Thread batch fashion.

5.3.3. CalcRegionValue

Regions in the simulation are bounded spaces in the radiant energy map relative to the owning creature. As the creature moves, the region moves with it. This also means that a region can be partially or completely outside of known space. The CalcRegionValue kernel handles the calculation of four statistics about each energy type-frequency in each region. These four values are minimum, maximum, total and count. Total is the summation of all values of a specific energy type-frequency in the region while count is simply a tally of how many valid spaces were used in the calculations.

Seeing as this kernel calculates statistics about regions, it makes sense that it divides the total number of regions to process among the different Blocks. As a result of the genetic evolution of the creatures, a region may be defined but ultimately be empty. The first step in the Block loop is to verify that the region being considered has any rows to check at all. The Block loop continues from here like most kernels with the coping of data from Global. Specifically, the list of relative offsets into the flattened radiant energy matrix is read for this region. The last step before starting the calculation is to initialize the local variables for the calculated values and initialize shared memory. Total and count are set to zero while minimum and maximum are set to the positive and negative of the maximum float value respectively. The last step of the setup is to have Warp zero initialize the shared memory with the default values.

Unlike most kernels, this kernel actually has an additional layer of division of work. Normally the work is divided by Blocks and then by Threads. In this kernel the work is divided by Blocks then by Warps and finally by Threads. The nature of the problem is what drives this. At the top level the regions are divided among the Blocks. Below that the flattened rows are divided among the Warps, each Warp handling a row at a time. Lastly, the Threads in the Warps handle one energy type-frequencies each. The difficulty with this method is the variable nature of the energy type-frequencies.

Since the number of energy type-frequencies is variable, simple methods would not work well. To explain, for every column space in a row there exists a full set of energy type-frequencies. The ideal case is when the number of energy type-frequencies just so happens to be the Warp width, 32. In this case each Thread handles an index in each row and nothing special has to be done. Obviously this is very unlikely to happen. There are two non-ideal cases, to small and too large. In the too small case the number of energy type-frequencies is less than the Warp width. A way to fix this is to just have each Warp process multiple columns at a time. Again this is unlikely to perfectly fill the Warp. The way around this is to just round down, meaning only put as many columns in the Warp as will completely fit. This may leave some wasted processing but it prevents Warp Divergence. In the kernel, the constant "WidthOfEnergySetsPerWarp" contains the width of the energy type-frequencies sets that can completely fit in the Warp.

The other issue is when the number of energy type-frequencies is greater than the Warp width. This basically means that each Warp cannot fully process a single column in a flatten row. The method used to resolve this was to create a virtual Warp width. The idea is to use as many Warps as is needed to fully process a column, meaning just multiply the Warp width by some positive nonzero integer to get a virtual Warp width that is sufficient. "WarpSizeNeededPerEnergySet" is the resulting virtual Warp width. This method is typically used in conjunction with the "too small" case given previously as doubling the Warp width will likely be more than is needed.

This kernel intern has several extra pieces of information related to the division by Warps and Energy type-frequencies. Warp width is the total number of virtual Warps in the kernel and Warp index is the index of each Thread relative to the start of their Warp. The last special value is energy index. This is the energy type-frequency that each Thread will consider. Each Thread can only handle one energy type-frequencies at a time but each energy type-frequency is likely handled by multiple Threads. Only handling one at a time is to keep the different energy type-frequencies separate.

The virtual Warp width also affects the information about the Warp. Warp ID is the ID of the Warp to which the Thread belongs. When using the virtual Warps, all Warps in the same virtual Warp share the same Warp ID.

The first step after the post initialization sync is to calculate the first and last valid column in this region. This value is relative to the creature's location and is bounded by the limits of the radiant energy map. Following this is two checks merged into one. This first is to check that the min is less than the max. If this is not true it means that there are no valid columns and the entire region can be skipped. The next check is if the current Thread has a valid Warp index. Since it is unlikely the virtual Warp will be completely filled, there will be Threads at the end that sit idle. Following this check is the Warp loop that assigns a subset of the flattened rows to each Warp. Now the row index can be retrieved and validated. Again, since the creature can move, there can be relative rows that are outside the bounds of the radiant energy map. These rows are just skipped and processing continues with the next row.

The innermost loop is what appears to be a Thread loop but is not a typical Thread loop. The difference is that this loop increments by the width of the energy type-frequencies processed by each virtual Warp, as opposed to the Thread width. The implication of this is that the inactive Threads would have actually been pointing to the data of the next cycle of the loop. As the Warps finished with their allotted set of rows, they atomically set their values into shared. Doing this required the creation of two additional methods based of the standard atomic operations [11]. Atomic Add was already available from CUDA, but using the template on their site, an additional two methods were created to do an Atomic minimum and maximum. Finally at the end, the values are copied back to Global before the next region is processed.

5.3.4. CalcNodeCharge

The processing of the Neural Networks for each creature was divided into two kernels for the sake of Shared memory capacity. The first is the CalcNodeCharge kernel that calculates the new node charge for each node relative to the incoming links to it. Being about the creatures, this kernel again divides the creatures among the different Blocks to be processed. Before any calculations are done in this kernel, each Block first checks that the creature it is processing has any nodes. A node-less creature can come from evolution but is also one of the marks of an uninitialized creature. In the initialization step, the old node charges and activations for the creature are copied from Global to Shared. Like with other kernels that process nodes, each Thread handles a subset of the creature's nodes.

For each node, the first step is to verify that the node even has links. Each creature has an array of structures containing the information about the links for each node. The structure of this list has been designed to improve performance but in turn is slightly complicated. The list starts with a single entry for every node even if the node has no incoming links. Additional links after the first are stored in a packed collated form after the set of first links. This means that all the second links for nodes that have second links appear in order and without gaps after the list of first links and before any third links. The third links follow and the pattern repeats until all links have been encoded. This structure was chosen to improve memory coalescence when retrieving the data from Global. Several tests were done with variations of different storage patterns. Each set of links being in order, sequential and pitched yielded the best performance. In this arrangement missing links are skipped and the next appropriate link on that level is instead written to that location. The pitching means that each level of links starts on a Global memory index. This intern means that there is wasted Global memory in this structure. This was deemed acceptable for the performance increase.

Starting with the first link nodes, the kernel checks that the linked node ID is less than the total number of nodes in the creature. This check might appear to be a simple safeguard but actually serves an important purpose. Because of the encoding structure every nodes must have a first link even if it has no links. This fake first link is marked as invalid by setting its linked node to hex 0xFFFFFFFF. Assuming the first link is valid; the associated node is next checked for activation. If active, its scaled and offset charge is added to the local tally of the node charge.

The information about the links consists of four pieces of information. The first three, (linked node, scale offset), were just used. The last is the byte offset to the next node within the link information array. Byte offset must be stored because the array is pitched at the beginning of each link level. This means that the number of bytes to the next location might not be a multiple of the structure size. In the kernel, the pointer into this structure is also kept as a byte pointer and cast to the link information structure for this reason. Each thread will loop over this array, offsetting its pointer by the stored byte offset, until if finds a zero byte offset. This zero byte offset is used to mark a link as the last link. Each linked node found this way is checked for activation and added to the node charge tally if active. The last task for the Thread is to write the updated node charges back to Global.

Table 1: Activation Functions

Positive	Negative	Relative Value	Activation Function
Empty	Empty		Charge != 0
Empty	Not Empty		Charge <= Negative
Not Empty	Empty		Charge >= Positive
Not Empty	Not Empty	Positive > Negative	Charge >= Positive || Charge >= Positive
Not Empty	Not Empty	Positive < Negative	Charge >= Positive && Charge >= Positive
Not Empty	Not Empty	Positive = Negative	Charge = Positive

5.3.5. CalcNodeActive

The second step in processing the neural network is to take the calculated charges and determine if the nodes are active. This is handled by the CalcNodeActive kernel. This kernel does not have any initialization before the Block Thread which like in the first step, assigns a subset of the creatures to each Block. Again, a check is made to make sure the creature has any nodes to process before continuing. The values copied from Global are the energy levels of each node for the current creature and the energy cost of each node to activate.

After a sync, the Thread loop assigns a subset of the nodes to each Thread in a similar manner to the first step. For each node individually, the activation costs are checked to make sure the node is even able to activate. This check is actually not completely accurate. Later in the kernel the activation cost of the activated nodes will be deducted. The deduction itself is atomic but is not synchronized with this early check. The consequence is that some nodes might activate that should not be able to activate. This was deemed an acceptable consequence since the cost to activate will still be subtracted. The only possible consequence is that a creature kills itself activating a node it should not be able to. This death is not a problem in the broader view of the simulation.

The main reason to pre-check for the nodes' ability to activate is to be able to skip the activation functions for any nodes that do not need them processed. The reason this is such a concern is due to the activation functions. Each node has two On Charges defined for it. Based on the values and relative values of these two On Charges, different activation functions are used. This leads to a lot of unavoidable Thread divergence. Luckily, the divergence is limited to only a short section and only has six distinct paths. So assuming the node can activate, what follows is a set of six if-else statements that choose the different activation functions. These different cases are listed in Table 1.

These six activation functions allow for a variety of behavior and even include band filters. The next step is to check if the node is active and subtract the activation cost if it is. Lastly, regardless of whether the node could activate or not, the new activation state is written back to Global before going on to the next node. The last step before moving on to the next creature is to write the new creature energy back to Global with the node activation costs subtracted from it.

5.3.6. ResolveActions

The ResolveActions kernel is the most varied and branching of the kernels. This is because this kernel handles the processing of all the actions. The concern with this kernel is Warp divergence due to multiple nodes being active. The counter to this is the assumption that for a Warp, the number of active nodes will be minimal so the Warp divergence of the kernel will be minimal. Ultimately, Warp divergence for this kernel is considered an unavoidable consequence of having multiple different actions.

Like other kernels dealing with node processing, this kernel assigns a subset of the creatures to each Block and a subset of that creature's nodes to each Thread. Again, the Block loop starts with checking if the creature has any nodes to process in the first place. Following this is the initialization step for that creature. Since this kernel deals with all actions, it has to copy a lot of different data from Global. Specifically it needs: creature energy, region values, action energy scale and action magnitude vector. Action energy scale is the energy type-frequency parameters of an action while action magnitude vector are the dimensional parameters of the action. The actual meaning of the parameters varies for each action. The last step of initialization is to zero out the shared memory holding the energy delta percentage per region and the kinetic energy impulse being applied to the creature.

The Thread loop comes next after the post initialization sync. Again each Thread in the Thread loop handles a subset of the nodes. After checking if the node is active, the flow of the kernel is split by a switch statement on the action type. This is the point of Warp divergence. Unfortunately, any active nodes within the same Warp are very likely to traverse this region of the code in a sequential manner.

The first set of actions are the sensing actions and they all behave in a very similar manner. For each energy type-frequency they add a scaled value to the node charge. What this value is depends on the action type. The scale used is the value stored in the action's energy parameters. The values comes from either the creature's own energy in the case of "SenseSelf" or a region for all the other cases. For

"SenseSelf" specifically, only energy type and not energy frequency is considered as the creatures only store frequency-less energy types. The other six sensing actions calculate the value they need from the region values: minimum, maximum, total, count. As a reminder, total is the sum of all spaces in the region of a specific type-frequency and count is the number of locations in the region. The six actions calculate their value to be used in the following ways: "SenseAverage" total / count, "SenseAbsoluteMax" maximum, "SenseAbsoluteMin" minimum, "SenseRelativeMax" maximum - total / count, "SenseRelativeMin" minimum - total / count, "SenseMaxMinDelta" maximum - minimum.

The final two actions are Absorb and Move. Absorb is used to absorb energy from the radiant energy map within the specified region. Within this action each Thread has to handle the processing of each energy type-frequency by itself. In this action the energy parameters are the percentage of energy to try to absorb for a particular energy type-frequency. Only the energy type-frequencies with significant absorption amounts are considered by this action. Specifically, this means greater than some set float value epsilon. Two things must be done for each of these energy type-frequencies. First, the percentage absorbed from the region must be atomically added to the total percentage absorb for that region. This is done locally in shared to minimize the cost of the atomic operation.

The second step is to add the absorbed energy to the creature. Since Absorb is region based, the "total" region value can be used to calculate the amount absorbed. A potential conflict with this method is that two creatures could be trying to absorb energy from two overlapping regions. This is why the percentage is stored for updating the radiant energy map. This percentage has the potential of getting to over 100% and driving the radiant energy negative. This is actually not a bad thing and will just be reflected in the region values for any effected region in the next cycle.

The last thing to consider with absorption is that it is not 100% efficient. In this simulation, the slower you absorb energy the more efficiently you can absorb it. The idea is that a creature can try and absorb 100% of the energy but will only get 50% of the energy. The remaining 50% is lost. The consequence is now they have completely depleted the storage of energy in that region. The efficiency was based on the idea that you have diminishing returns the more energy you try to absorb. The equation for the amount of energy absorbed is total * scale * (100% - (0.5 * scale)). This equation behaves in a way where the percentage of energy you lose is half the percentage you gathered. This loss percentage is applied to the energy absorbed and not the total energy. For instance if you were trying to gather 100% you would loss 50% of the 100% you gather. A better example would be trying to gather 50%. In this case you lose 25% of the 50% you gathered or 12.5% of the total energy. This also has the implication that if you try and gather 200% you will lose all the energy you gathered while setting a very negative value in the radiant energy map. Even further past 200% and you are actually losing energy when you try to absorb. The last step of this action is to atomically add the absorbed energy to the creature energy stored in Shared.

The last action is Move. What Move does is transfer creature energy into kinetic energy. For this action the energy parameters contain the quantity of energy of each type to transfer and the dimensional parameters contain the unit vector of which direction to move. The first step is to remove the energy to be transferred from the creature. This action must be done atomically and only if there is sufficient energy. Because of this, a specialized version of the atomic operations was done in place. It was not made into a function since it is not used anywhere else and has very special behavior. The main difference between this atomic subtraction and others is that the Thread will skip the operation if the creature has insufficient energy. The last step is to apply the total transferred energy to the individual dimensions based on the given unit vector in the actions dimensional parameters. This step is skipped for trivial amounts of transferred energy.

After the action for the node has been processed, the only cleanup work is copying the new node charge back to Global. The sense actions add their sensed values to the already activated node. While this does not affect the activation of the node in this cycle, it will affect the strength of the signal to linked nodes in the next cycle. The processing of the creature as a whole finishes by updating the creature energy, the region energy delta percentages and the creature kinetic energy. The last step is to calculate the creature's radius and mass. These values are used by several kernels later in the cycle but are calculated here. Stored energy it used to calculate both the mass and radius of the creature. Additionally, the number of dimensions effects how the radius is calculated as it is based on the spheroid volume needed to contain the creature's energy at some set density.

5.3.7. ApplyRegionEnergyDelta

Updating the radiant energy map after energy has been absorbed by a creature is actually a very difficult task to do in parallel. The problem is this task requires different types of work divisions at different steps. Specifically, it requires a division by creature and node to determine what percentage is being absorbed per region. This absorption rate per region then needs to be applied to the radiant energy map, which requires a division by region. The problem with this is that the change in energy is represented as a percentage of the initial value. If the changes were applied sequentially and two regions overlapped, then the absorbed energy from the radiant energy map would be less as one of the two would see a lower initial value. The solution that was chosen after much consideration was to create a copy of

the radiant energy map that holds the total percentage change to be applied to each region. This percentage change is the same value that was used in the CalcRadiantEnergy kernel.

For this problem the ResolveActions kernel handles the generation of the absorption percentage region and the CalcRadiantEnergy kernel will handle applying the percentage changes to the radiant energy map. This kernel handles the conventions of the region percentage energy change to the percentage energy changed map. Like CalcRegionValue this is a kernel that processes regions. Again it starts by assigning a subset of the regions to each Block. If the region has any rows to consider, it then proceeds to copy the region energy delta percentages from Global to Shared. To save some work, the read values are pre-checked for non-zero values. If all the percentage change values are zero then there is nothing that needs to be done for this region and it can be skipped.

Assuming that there are percentage energy deltas to process, the kernel starts the region handling in a similar manner to CalcRegionValue. Again, the first step is to copy the relative offsets to the needed rows for the region from Global. Next, check if the current Thread's energy type-frequency has a delta energy percentage associated with it. Just like the region case, skip any energy type-frequencies that do not need to be processed. Following this is again the calculation of the minimum and maximum value columns and the check to see if there are any valid columns to consider. The unneeded Threads are again skipping like before. Next, check if the row is valid and if it is, then atomically add the percentage energy delta to the Global energy map for radiant energy delta percentage. This action is again a necessary inefficiency. Atomic operations with Shared are costly to begin with. When they are made with Global they become something to be avoided. Unfortunately, a better method of dealing with overlapping region updates was not able to be found so the added cost was just taken as unavoidable. Since the new values are written as part of the inner loop, this kernel has no final update section.

5.3.8. ResolveCollision

The ResolveCollision kernel deals with the detection and physics of collisions between creatures. Ultimately its only purpose is to calculate the total change in momentum due to collisions to be later used by the ResolveMovement kernel. Before even defining any of the variables, a check needs to be made that there are more than one creature. A creature cannot collide with itself so any work in this kernel when there are less than two creatures is pointless. Before the Block loop, creature positions and radii are copied from Global to Local. This is done since these values will be accessed numerous times by many Threads throughout the execution. As can be expected, the main Block loop for this kernel has each Block processing a subset of the creatures. Since the radius was already copied from Global it can be used to check for uninitialized creatures. A zero radius is one of the tags for an uninitialized creature.

One of the difficulties of this kernel turned out to be figuring out how to evenly divide up the work among the different Blocks. The issue is that in order to check for collisions, one needs to calculate the distance between every pair of creatures. There were two ways considered initially. First, have every creature check every other creature for collisions. This method is the simplest but introduces wasted computations. With it the distance between each creature ends up being calculated twice each cycle. The other initial idea was to have each creature only check for collisions against creatures with IDs higher than itself. The problem with this method is that it off balances the load of the calculations to the lower ID creatures. Since the task is not completed until all calculations are done, this leaves the Blocks related to the higher ID creatures idle. Ultimately, a good solution was found via doing several experiments in Excel with 2D tables.

The solution was to have each creature check some number of the next creatures following it in IDs. The IDs to check would wrap around at the end of the list. The number of following creatures was different depending on whether the creature was in the lower or upper half of the IDs. For the lower half, the number checked was half the total number of creatures rounded up to the next integer. For the upper half, the value was instead rounded down. This split allowed every possible unique combination of creatures to be considered only once and for the work to be divided nearly evenly among the Blocks.

The Thread loop for this kernel uses the subset of the creatures figured out in the previous step. Firstly, each Thread finds the square magnitude of the distance between that Block's creature and the creature the Thread is checking. If this is less than the total of the two creatures radii squared, the creatures are considered to be in collision. After taking the square root of the squared distance found early, the magnitude of the change in moment is found. Originally this calculation was far more complex.

The initial plan was to not allow the creatures to overlap. This means that if two creatures were found to be overlapping, then they would be rewound to the exact moment of collision. The changes in momentum due to a partially elastic collision would be calculated at this time and then the creatures would be fast forwarded to where they should have been after the collision. The problem with this method is, while it can work in a sequential system, it is very difficult in a parallel system. Ultimately, the problem lies in having to move the creatures to resolve the collisions. Every time a creature moves, it has the potential to create new collisions. To solve this system with no overlap would require a complex system of equations or a numeric method to converge in on a solution. The problem is that both of these are intensely time consuming and extremely difficult to implement in parallel. These methods also have the

added risk of having no solution or never converging. Due to complexity and both high and uncertain cost, it was decided to utilize a less accurate but more reliable method.

The method chosen was to use a simple spring force in the direction of the vector between the creatures. Back in the collision kernel, when this is calculated, it is also pre-divided by the magnitude of the vector between the colliding creatures. This is done so that later on the raw distance between the creatures can be used to find the components in the individual dimensions as opposed to having to calculate a unit vector. Lastly, this delta momentum is atomically added and subtracted from the two creature's tally of momentum changes due to collisions.

When handling the collision, a special case must also be considered. This special case is when the two creatures are nearly or exactly at the same position. Having a total this close to zero causes rounding errors to be magnified and generally destabilizes the calculations. The solution is to use the total of the two radii as the overlap and the square root of the number of dimensions as the vector magnitude. This effectively places the vector between the two creatures in the positive direction for every dimension. This value is then added and subtracted from the creature's tally of momentum changes due to collisions.

5.3.9. ResolveMovement

The Resolve Movement kernel is the primary physics processor of the simulation. Its primary purpose is to calculate the new creature positions and momentums. The Block division for this kernel is over the set of the creatures. Before the Block loop, the size of each dimension is loaded into shared to be used later when checking for wall collisions. Dimension sizes are constant so they should only be loaded once before the Block loop. The first thing checked is if the creature has a positive nonzero radius. A zero radius is one of the ways creatures are marked as uninitialized. Following this is the loading of the old creature velocity and current creature energy. These will be needed several times so they are worth preloading.

After the setup stage, several values will need to be calculated that unfortunately must be calculated sequentially by Thread zero. Specifically, the total stored energy and velocity magnitude must be calculated. This is a reduction operation performed over the number of energy types and the number of dimensions respectively. The size of both of these reductions will tend to be small as the hardware is unlikely to be able to handle larger values. For this reason, more parallel techniques were not used due to their overhead. Lastly, for nontrivial velocity magnitude, the forces opposing velocity are calculated. Specifically, the change in momentum due to friction and drag is calculated. This value is also preemptively divided by the velocity magnitude. This value is originally a vector magnitude and thus must be multiplied by the unit vector of velocity to get the individual contributions in each dimension. By preemptively dividing it by the velocity magnitude, an extra operation over each dimension is avoided later.

With the sequential tasks completed, the primary Thread loop can run. This Thread loop divides the work up over the individual dimensions in the simulation. The change in momentum due to velocity opposing forces is applied first. Special care has to be taken when subtracting this value, because it cannot be allowed to change the sign of the momentum. Momentum is set to zero in the cases where the sign would have changed. Next, the added kinetic energy from the Move action is added. Special care again has to be taken here to preserve the signs of the vectors through the square and square root operations.

Collisions with the walls of the environment are handled next. This calculation was originally being done with more scientifically correct equations. Ultimately these equations proved to be too rigid to successfully use at the needed time scales. The alternate, albeit less scientifically correct, was to use a simple spring constant and allow the creature to penetrate the wall. The original method did not allow penetration. This ultimately was what caused the method to fail as resolving one collision could cause others and so on. This turned the problem into a system of equations that was far more complicated than the simple physics needed in this simulation.

Finally, the new momentum can be calculated along with the damage sustained from any collisions with other creatures or the walls. For damage, a base defense was defined that subtracts from any damage taken down to a minimum of zero. In later revisions, defense will be a whole system that can be actively controlled through actions. The last values to be calculated are the new velocity and the new position. Following this, the new values are copied back to Global or Shared Memory and the impulses from other kernels are cleared.

Now out of the Thread loop, the total damage the creature sustained must be tallied by Thread zero. If there was any damage sustained, the taken damage is divided among the creature's different energy types relative to the quantity of each energy type. Finally, the new row and column location of the creature in the global radiant energy map must be calculated for the new creature position.

5.3.10. CheckReproductionAndDeath

This kernel ultimately acts as an event manager. Its primary purpose is to signal the CPU whenever a death or reproduction event happens. Its secondary but ultimately more frequent task is to deduct the maintenance energy from each creature every cycle and check if that has caused the creature to die. Again, the Blocks in this kernel each handle a subset of the total creatures. The first task in the Block loop is to check that the creature has been initialized. A creature status of NULL is used in this case as another flag marking uninitialized creatures that should be skipped. The zero Thread then initializes the shared memory Booleans for

death and reproduction. These values are only ever set true from this point forward so any potential write conflicts from Threads will not cause any problems. Simply put, this means atomic operations are not needed on these shared variables.

Following this is the first Thread loop of the kernel. In this loop each Thread handles an energy type, checking if the current creature has sufficient energy to pay the maintenance cost. If it does, the cost is deducted and stored back to Global. Otherwise, the creature is marked as dead in Shared. After syncing the Threads, any creatures that have not died are further checked for reproductions or voluntary death. This time the Thread loop is over the creature's nodes and begins by checking that the creature has not chosen to die. For any active nodes, the death and reproduction tags are set if the node's action is of the corresponding type. Lastly, Thread zero updates the creature's status in Global.

6. Experimental Results

While the simulation works, it ultimately did not do quite what it had been designed to do. The evolution of the creatures turned out to be too much for the hardware to handle. Initially, the assumption had been made that the frequency of reproduction and death would be infrequent. This proved to be wrong. In the actual simulation, an actionable event happened nearly every cycle once the creatures became numerous. These events transfer control back to the CPU for the processing of reproduction and death events. This processing is sequential and also involves memory allocation and deallocation in both the CPU and GPU. This block of code ended up being the bottle neck of the simulation.

There are several ways this bottle neck could be mitigated in future development. Firstly, reproduction could somehow be made more costly to the creatures. This should reduce the frequency of the reproduction code. To lessen the memory management, all possibly needed memory could be pre-allocated in fixed arrays. This would not be too much of a problem to do as the simulation already pre-checks the memory limits for the given simulation against the GPU. Lowering the maximum number of creatures or nodes per creature would also help but at the cost of further limiting what the simulation can do. The last option, which is the hardest and very impractical, is to somehow move the reproduction operations inside the GPU. This method would necessitate the constant memory allocation method mentioned previously. The main problem is that the DNA representation of the creatures would have to be stored and processed into the low level representation completely within the GPU. This system would likely consume a significant portion of the Global memory.

Even with the problems that limited the simulation, it still produced some interesting results. The early runs were mostly focused in finding an initial state that would run for the full duration of the simulation. The first simulations nearly always resulted in extinction. In other words, the environment had not been made conducive to life. This was solved by tweaking the ambient energy to give them more food. This eventually led to too much food and the costs of nodes and links were increased to balance it.

The next source of simulation failure was collision death. This was caused by the constraints associated with motion being set such that motion was too easy. This result showed itself in a cascade like manner. The simulation would run normally until it reached a critical number of creatures. Afterwards one collision would cascade into others and at the end nearly all creatures would be dead. The solution to this was to make movement harder and collisions not as harsh.

At this points the world had been tuned enough to keep the creatures alive so evolution had time to work. The first noticeable evolution was actuality sterility. Since the initial creature was designed to be the bear minimum to survive, it was susceptible to mutations rendering its children sterile. This problem was recognized from the drastic decrease in cycle time and the nearly constant creature population. The solution to this was to artificially inject reproduction back into the population. If a full batch of cycles was executed without a reproduction event happening, a reproduce would be forced between some random creature and the initial creature. This proved to be very effective at preventing the early simulation sterility problem.

The second evolution of note actually allowed the creatures to become immortal via exploiting a bug in the simulation. The exact mechanism of how they managed to do it was not found but what the creatures had evolved to do was have an energy of NaN. Energy in the creatures is stored as a floating point number which have several special values. One of these special values is Not a Number (NaN). The problem with NaN is that any comparison operation against a NaN will always return false. The bug was in the logic for detecting a dead creature. The check was testing if the creature had insufficient energy to pay the maintenance cost. If this was true the creature was marked dead. The problem is that with NaN, this comparison could never be true so even though the creature was dead it could not die. Even worse, this change would be passed on to its children. The solution was to reverse the comparison, so that it was instead checked if there was enough energy to pay the maintenance cost. If this was false then the creature would be marked as dead. This solution allowed any creatures that had developed a NaN energy to be removed from the population.

7. Conclusion

In the end, the simulation works, just not as well as expected. At the beginning, the assumption was made that reproduction would be infrequent. This unfortunately ended

up being incorrect. The result was that the sequential reproduction algorithm consumed the majority of the processing time. This bottle neck robbed any processing efficiency gained from using CUDA as the GPU sat idle. Even with this bottle neck the simulation still managed to show interesting results. The most notable being the creatures evolving to find a bug in the system that allowed them to become immortal. Future work for this simulation would mainly involve solving the reproduction bottle neck. This bottle neck made very high cycle count runs impractical, which limited the system's time in which to evolve. Ultimately the simulation did show creatures evolving in the limited cycles it could practically be run for. With better hardware and a better reproduce method, this simulation has the potential to do much more.

8. REFERENCES

[1] NVIDIA, "What is CUDA," [Online]. Available: https://developer.nvidia.com/what-cuda. [Accessed 2013].

[2] 3D Game Engine Programming, "CUDA Memory Model," [Online]. Available: http://3dgep.com/?p=2012. [Accessed 2013].

[3] M. Dorigoa and C. Blum, "Ant colony optimization theory: A survey," *Theoretical Computer Science,* vol. 344, no. 2–3, p. 243–278, 2005.

[4] A. C. S. Raoa, D. Somayajulub, H. Bankaa and R. Chaturvedia, "Outlier Detection in Microarray Data Using Hybrid Evolutionary Algorithm," *Procedia Technology,* vol. 6, p. 291–298, 2012.

[5] C. Blum and V. Schmid, "Solving the 2D Bin Packing Problem by Means of a Hybrid Evolutionary Algorithm," *Procedia Computer Science,* vol. 18, p. 899–908, 2013.

[6] A. Blanco, M. Delgado and M. C. Pegalajar, "A genetic algorithm to obtain the optimal recurrent neural network," *International Journal of Approximate Reasoning,* vol. 23, no. 1, p. 67–83, 2000.

[7] S.-K. Oha and W. Pedrycz, "The design of self-organizing neural networks based on PNs and FPNs with the aid of genetic optimization and extended GMDH method," *International Journal of Approximate Reasoning,* vol. 43, no. 1, p. 26–58, 2006.

[8] L. M. Schmitt, "Theory of genetic algorithms," *Theoretical Computer Science,* vol. 259, no. 1–2, p. 1–61, 2001.

[9] L. M. Schmitt, C. L. Nehaniv and R. H. Fujii, "Linear analysis of genetic algorithms," *Theoretical Computer Science,* vol. 200, no. 1–2, p. 101–134, 1998.

[10] J. Siegel, J. Ributzka and X. Li, "Memory Layout Optimization Techniques for Large Data Structures on CUDA," [Online]. Available: http://www.eecis.udel.edu/~mpellegr/eleg662-09s/. [Accessed 2013].

[11] NVIDIA, "CUDA C Programming Guide," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 2013].