GPU Based Large Scale Multi-Agent Crowd Simulation and Path Planning

Luke S. Gusukuma

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Application

Yong Cao, Chair
R. Benjamin Knapp
Dane Webster

April 26, 2015
Blacksburg, Virginia

Keywords: Crowd Simulation, GPU, Roadmap, CUDA, Parallel Computing

GPU Based Large Scale Multi-Agent Crowd Simulation and Path Planning

Luke S. Gusukuma

ABSTRACT

Crowd simulation is used for many applications including (but not limited to) videogames, building planning, training simulators, and various virtual environment applications. Particularly, crowd simulation is most useful for when real life practices wouldn't be practical such as repetitively evacuating a building, testing the crowd flow for various building blue prints, placing law enforcers in actual crowd suppression circumstances, etc. In our work, we approach the fidelity to scalability problem of crowd simulation from two angles, a programmability angle, and a scalability angle, by creating new methodology building off of a struct of arrays approach and transforming it into an Object Oriented Struct of Arrays approach. While the design pattern itself is applied to crowd simulation in our work, the application of crowd simulation exemplifies the variety of applications for which the design pattern can be used.

Table of Contents

List of Figures

Chapter 1 Introduction

1.1 Motivation

One of the strongest motivating factors behind crowd simulation is to test for hypothetical situations. More specifically, these are hypothetical situations that either would involve a lot of man power to test, or would require large amounts of resources (e.g. a building still in its preliminary design stages). The wide range of applications includes (but is not limited to) training simulators, video games, movies, building planning, event planning, and evacuation planning. The application of crowd simulation most useful when the situation to be simulated would be infeasible in real-life. For example, constructing a building only to find out that a corridor is too narrow for appropriate throughput is expensive. Likewise, paying for hours to arrange guards in a building to observe how trafficking people in different situations would affect evacuation efficiency is also expensive. In cases like this, crowd simulation is both cheaper and more time efficient than playing out scenarios in real life.

Because of the number of useful applications for crowd simulation, it has developed into a relatively deep field of research. The research areas of crowd simulation itself can roughly be broken down into the following (not mutually exclusive) categories: modeling, performance, and analysis. Our research focuses on the modeling and performance aspects of crowd simulation.

As a general rule of thumb, when creating incredibly large crowd simulations, experts don't think of agent based simulation because of its inherently larger computational work load. Additionally, with regards to the GPU in particular, agent based crowd simulation remains largely unused due to difficulties in programming agent based simulations on the GPU because it doesn't work well with the GPU; typically, large scale crowd simulations on the GPU favor approaches that leverage inherent hardware features [1] [2] [3] [4] [5]. In our work, we break this notion with the Object Oriented Struct of Arrays (OOSoAs) design construct, a design pattern that simplifies complex formulations such as Wei's Autonomous Pedestrians [6] and Curtis' Way portals [7], which we use as case studies for the purposes of testing our design pattern. The results of our labor are two things. The first is a framework that balances high performance computing on the GPU with software engineering programmability using a variation on object oriented programming that runs more efficiently on the GPU and consequently, a crowd simulation on the scale of tens of thousands. The second thing, is that we explore the OOSoAs design construct as a tool to not only port complex heavily object oriented frameworks to the GPU, but also as a tool to simplify code analysis by reducing the complexities in analysis caused by poor memory access patterns.

1.2 Research Contributions

1.2.1 Contributions to General Programming for GPUs

Because GPUs aren't normally designed for general programming, there is a large sacrifice in programmability when trying to leverage the power of GPUs for non-graphics applications. Traditionally, GPU programming in the C/C++ environment is done using raw C code and a struct of arrays (SoA) management of data. While using a SoA approach increases efficiency due to the coalesced memory access patterns [8], the program suffers from organization issues when refactoring, rearranging, and writing code. As an opposition to performance in favor of programmability and higher level problem solving, Object-Oriented Programming (OOP) has come to the forefront as a strong methodology for creating readable/modular code for traditional programming. In CUDA 5 and 6, they created adaptations for using object oriented programming on the GPU [9], however performance of these objects is poor due to the inherent speed and access patterns of the GPU [8].

In our research, we have designed a framework that strikes a balance between the performance of a SoA management of data and object-oriented programming, creating a new hybrid software engineering design we call Object-Oriented Struct of Arrays (OOSoA). With the OOSoAs design pattern, we enable a natural design formulation of agent-based crowd simulation on the GPU, while maintaining both relatively strong performance as well as OOP programmability. This in turn, allows us to think of the

problem from a higher vantage point as each agent as an individual entity from the abse framework

Additionally, beyond just improving programmability of GPU programming, the OOSoAs design pattern also increases the accessibility of GPU programming to a broader audience by enabling object oriented programming on the GPU and simplifying dealing with the memory problems that arise with GPU programming.

1.3 Thesis Outline

A Literature review about recent research done on crowd simulation performance and modeling is presented in Chapter 2. Chapter 3 overviews the design of the crowd simulation we build. An analysis of the crowd simulation design and performance is contained in Chapter 4. Concluding remarks and future work are given in Chapter 5.

Chapter 2 Literature Review

Crowd simulation comes in a number of different styles ranging from more physics based simulations such as continuum dynamics simulations to more human-like agent-based simulation. There is a large breadth of crowd simulation varieties in this range, each with their own strengths and weaknesses regarding interactivity, scalability, realism, etc. However in general, crowd simulations tackles the problem from two points, macroscopic and microscopic modeling. Macroscopic modeling takes an approach of viewing the system holistically. Examples of this include many physics/holistically oriented options such as continuum and fluid dynamics that focus on the overall flow of agents. Typically, this allows for the system to handle impressively large numbers of agents interactively, but the individuality of each agent is lost. Microscopic modeling on the other hand focuses on the individuals of the system with the flow of agents emerging as a consequence of the interaction between individual agents. Typically, this approach allows each agent to be different and unique to a certain extent. However, because of the amount of calculations and the need to deal with each agent individually, microscopic modeling based approaches struggle to scale up to the size of crowds that macroscopic modeling allows.

Our research primarily falls under microscopic modeling, attempting to scale it up to larger magnitudes. We tackle the problem through the use of GPU programming using CUDA to scale a complex agent based behavioral model to new heights.

2.1 Simulation Scalability

2.1.1 GPU Accelerated Crowd Simulation

The use of the GPU for general programming has been around for a little more than a decade, with its beginnings for non-graphics usage starting up around 2001 to 2003 [10]. As far as using GPU programming for crowd simulation in particular, it seems to date back to about 2008 [4] [5].

Passos [4] used an agent based simulation like in our work; he accelerates crowd simulation on the GPU by implementing a grid/matrix based data structure that is mapped onto texture cache to handle sorting and nearest neighbor searching. However, the simulation implementation he uses is an implementation of Reynolds [11] flocking in 2D. Other approaches that are used and implemented on the GPU use macroscopic modeling such as continuum dynamics [5] [3].

The work of Shopf et al. [5] uses continuum dynamics from [12] to outline overall global path planning and a local collision avoidance structure, borrowing from [12], and using novel techniques to expand it onto the GPU. It starts with a low resolution potential field for global navigation, solving a simplified eikonal equation in parallel [13]. This plans the global paths for several agents simultaneously. At the local level, they use a velocity obstacle approach [14]. For efficient queries they used GPU binning with a combination of texture memory and depth buffers to optimize neighbor finding to achieve optimal memory access patterns for neighbor finding. They manage to get around 65,000

agents in real time. Ultimately, it suffers from the same weakness that most other macro modeling approaches suffer from, which is the loss of individuality of the agents.

Grid based approaches such as the previously mentioned continuum dynamics approaches have received much popularity with application to the GPU due to the inherent spatial locality of texture memory working well with the grid based approaches [2].

The work of Demeulemeester et al. [3] builds off of Shopf et al.'s work further. To build off this work, they implement a refinement to Shopf's work. Before solving the eikonal equation, they implement a coarse A* based solver to determine a set of regions of interest for the path planning, reducing the amount of necessary computation needed for the eikonal equation and produce a faster simulation reaching magnitudes of 100,000 agents.

A more closely related work, ClearPath, is one of the more comparable works in existence relative to this work in terms of scale and magnitude [15]. ClearPath is an agent based simulation like ours, using a velocity obstacles methodology with making some assumptions to make it more parallelizable. However, with Clear Path, they use a Larrabee simulator, which simulates a hybrid CPU-GPU architecture so it's difficult to make a comparison with this work regarding scalability as the architecture it uses is completely different and isn't actually produced by Intel.

Our approach uses GPU programming to run an agent based crowd simulation at interactive levels. Specifically, we model our crowd simulation off of Shao Wei's [16] framework, implemented using OOSoAs on the GPU. With our model, even with the complex cognitive modeling structure and behavior based agents (microscopic modeling), we reach competitive levels with conventional implementations of macroscopic models of crowd simulation such as Treuille's Continuum Crowds [12]. Our simulation performance lies between CPU based physics models and GPU-based physics model. To our knowledge, our system is the first to be full blown implementation of a completely agent based behavioral model with cognitive modeling on the scale of tens of thousands.

2.2 Agent Based Modeling

In this section, we go over crowd simulation work on the agent based side that relates to our work. Starting with this is Reynolds [11] seminal 1987 work on behavior driven agent based crowd simulation using boids. Since then there has been a number of agent based approaches. However, the agent based approaches that stand out most recently are velocity obstacle approaches [17] [18].

The core of the simulation that we used is largely based on Wei's Autonomous Pedestrians [16] [6], is also an excellent example of the cognitive modeling for agent based simulation. Wei's model had two major layers, which were the cognitive modeling of the agent and the motor control behaviors of the agent. In the cognitive modeling

aspect, Wei's model was based on a goal oriented motivational structure that broke into sub-goals in a stack based approach, mimicking a user's memory. At a lower level, Wei's mode uses a series of reactive, human like behaviors for basic motor control such as "stop if an agent is too near", "go with people that are moving in similar directions", and other similar very descriptive behaviors, with some situationally specific behaviors for various complex situations. In our work, we use the goal decomposition and the reactive behavior aspects of his work in our implementation as our starting point. The reason for this was to define a framework on the GPU that had a relatively high granularity in the control of the agents.

Another closely related work (previously mentioned) is Curtis' Way Portals [7]. In his work, he converted typical roadmaps into line segments that the agent would approach rather than individual points like in typical roadmaps. The advantage to this approach was that agents would use more of the available open space. Additionally, this type of approach coincides more with the idea that people approach areas rather than points when navigating a space. This approach lends itself to a higher granularity of control of individual agents. In our simulation, we use a more detailed variant on Curtis Way Portals formulation, which also adds to the complexity to our system as a whole.

Chapter 3 Crowd Simulation Design

In this section we go over our design from two major aspects, the overall design of the crowd simulation, the representation of the world itself, and our construction of the OOSoA.

3.1 Crowd Simulation Design

As we mentioned before, our crowd simulation model was largely based off of Wei's model, mainly borrowing the reactive behaviors and the cognitive modeling of his system, but with a focus on the design and object oriented programming of the simulation. To our design, there are two major pieces, the agent with its cognitive modeling and behavioral motor controls, and the navigational/path planning aspects, each of which we will go in-depth.

3.1.1 Structure Overview

Before discussing the different components of the system in-depth and discussing the system holistically in-depth, we'll start with a brief overview of the structure of the system with enough of the necessary detail to start to talk about the individual components in detail. In a later section, we'll address the system as a whole.

In our system the Agent is an object, as with any object oriented agent based simulation. Each agent itself contains its own cognitive model consisting of a stack of

goals (which will be elaborated later), and an ability to act, both of which will be elaborated later. Additionally, each agent has various fields that represent different attributes of the agent in addition to the goal stack, such as its current position, velocity, etc. The main body of the simulation can be seen in Figure 1.

```
inline void bodySim(int i){
    Agent* curAgent = World::getAgent(i);
    Goal* currentGoal = curAgent->getCurrentGoal();

    Action* theAct;
    bool wasExecuted = currentGoal != NULL;
    //Check if the agent has any goals in memory
    if(wasExecuted){
        theAct = currentGoal->getNextAction();
        //Set up conditions for acting
        theAct->execute();
    }
    //If the goal hasn't been completed
    if(!curAgent->currentGoalComplete()){
        //run agent's unique behaviors
        curAgent->act();
    }
    if(curAgent->isMoving()){
        curAgent->position() = curAgent->position() + (curAgent->vel())*World::getDeltaT();
    }
}
```

Figure 1: Crowd Simulation Body device code

Navigation is based on a goal oriented motivation structure. In the most basic case, the goal will be to "Go to Region X." In the "Go to Region" case, the decomposition of goals leads to a movement *Action* that leads the agent towards a particular point in space, which in turn creates a desired velocity given the agent's desired speed. In a typical roadmap scenario, the point would be an established point on the roadmap, however, in our system, these points are derived from the curve based roadmaps, in objects that we call "Portals." Depending on the function that defines the portal (a "Get point on portal" function), the agent will identify a point on the portal

towards which to travel. The details behind portals and goals will also be elaborated in a later section.

The rendering itself is done in a completely different unit which gets information from the simulation and isn't included in our design beyond a module to output different ways to render the orientation and positions of the agents.

3.1.2 Agent modeling

In this section we describe how we model the agents, and explain some of the reasoning behind specific design choices.

3.1.2.1 Agent Cognitive modeling of Agent

For this part of the system, we used a stack implementation of the goals which decomposed into other goals. In the original design [16] [6], goals automatically split themselves into more goals and placed themselves on the stack. However, in the case of the GPU, such frivolous memory access (constantly reading and writing to global memory whenever part of a goal was complete) and results in frequent removal and adding of goals to the stack. To remedy this we remodeled how goals were placed on the stack as well as how goals were decomposed.

In our model, goals are considered an object, which are composed of other goal objects. This construction creates a sequence of goals. At the base of this tree of goals are leaf nodes that are action objects, which translate into an action that an agent can take.

An example of this structure is displayed in Figure 2, with an arrow representing the order in which goals are executed.

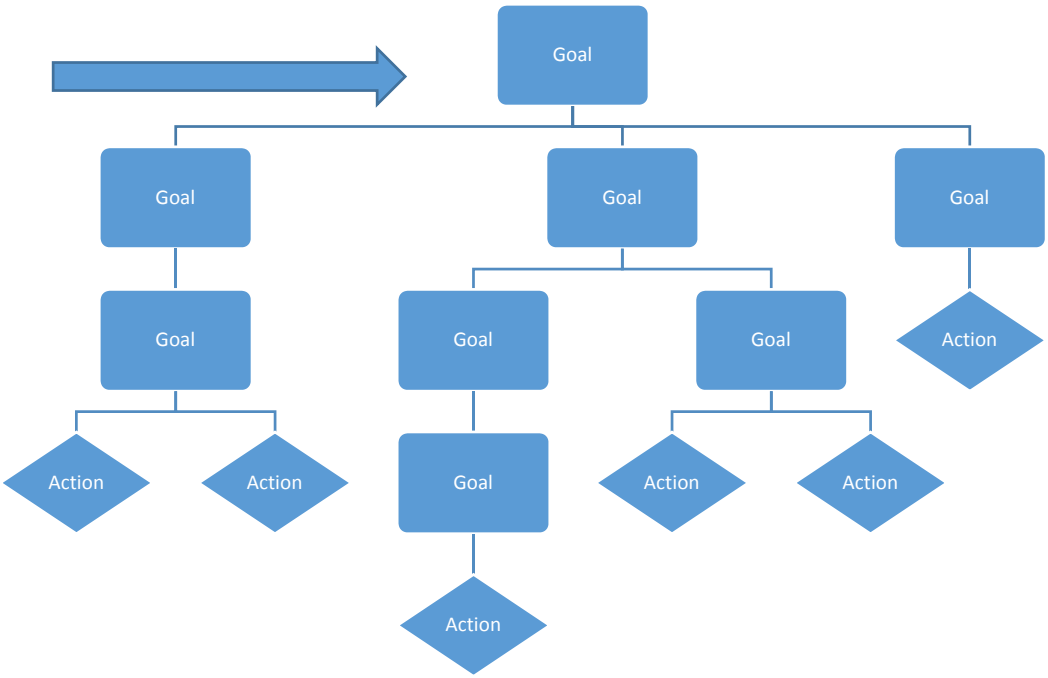

Figure 2: Basic Goal Construction

The action unit itself represents the most basic actions that an agent can take, which also denotes leaf nodes of a particular goal. In our system, each agent has a stack of these goals. As shown in Figure 2, the goal object is queried for its next action by calling a method to get its next action. In this fashion, the entire goal can be kept on the stack as opposed to constantly writing new goals to the stack once some goal is completed, reducing the number of memory writes.

In our system, the goal is the crux of the cognitive modeling aspect. Each goal has a pointer to the agent that owns it. Depending on the contents of the goal, the agent surveys the world for information relevant to the action being executed, updating the

agent's perception of the world. Finally, the agent's act method is called which applies the reactive behaviors and actually moves the agent, whose implementation is discussed in the next section.

3.1.2.2 Agent Motor Control

The motor control of the agents consists of the reactive behaviors that were lifted from Wei's autonomous pedestrians [6] [16]. Quickly recapping these rules are static obstacle avoidance, static obstacle avoidance in a complex turn, maintaining separation in a moving crowd, avoid incoming pedestrians, avoid dangerously close pedestrians, and verify new directions relative to obstacles. In our program we maintained relatively faithful to these behavior patterns, however, the design of the implementation of these on the GPU is worth mentioning. Global memory access is one of the slowest aspects of GPU programming, particularly. Because of the memory access patterns of the GPU, it was more efficient to take a summative approach to the agent avoidance behaviors rather than to take them as is from [6] [16]. To maintain different orderings of the behavior, it was necessary to observe all nearby agents at first, and calculate their overall effect with regards to velocity, and then applying them to the agent in a summative manner rather than calculating each individually. Because of this approach, it isn't quite feasible to implement the behaviors exactly as prescribed in [6] [16] on the GPU, but we can get something relatively close. The object avoidance behaviors on the other hand, could still be maintained relatively well, even when transferred to the GPU.

The major meaning of the change to a summative methods was that velocity comparisons between steps in the reactive behaviors in determining "most similar direction" were calculated based on the velocity at the beginning of that step rather than changing as each reactive behavior was applied. In testing both versions of this code, we found little difference in the resulting movements and behaviors.

3.2 Navigation of Agents

The navigation of agents in our simulation is similar to typical agent based crowd simulations in many aspects, they navigate to a series of points defined on a roadmap, which leads the agent to its desired destination. However, our navigational structure is conceived at a higher level than simply navigating to various points in the space. In our simulation, we use a variation on Way Portals [7].

3.2.2 Way Portals Variation

In this section we go into the details of the modification on Way portals that we use in our simulation

3.2.2.1 Space Partitioning

In our formulation, we manually partitioned the space into topological regions as opposed to manipulating a roadmap itself. The purpose of traditional roadmaps in crowd simulation are to navigate around objects that agents are supposed to avoid and create built in paths that agents should travel. Our formulation split the difference between the

pure topological division of the area and the point to point roadmaps that are typically used. That is, our formulation on Way Portals partitions the space to the level of occluding corners (see Figure 3 black line), but leave smaller obstacle avoidance to the behaviors of the agents.

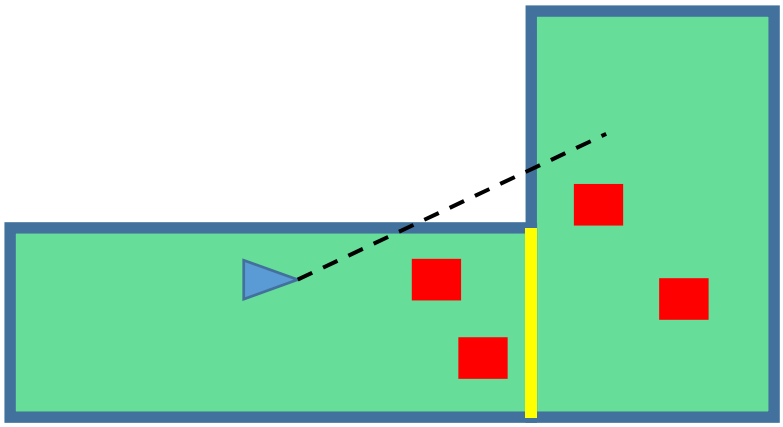

Figure 3 Roadmap visualization: Yellow represents a curve or "portal", red represents smaller obstacles not accounted for by the roadmap, the black dotted line shows an example of an occluding corner, and the triangle represents an agent.

By placing the roadmap at a higher level of abstraction, we end up with a more intuitive navigational structure than before. As a way to explain this intuition, from the perspective of a person, walls serve more as a boundary of movement rather than something to avoid. This thought pattern is reflected in how the boundaries, which we refer to as portals from here on out. On the other hand, obstacles, such as the red boxes in Figure 3, are the types of objects that people in real life would actively walk around. In true object oriented style, we use this new formulation as an increase in the complexity of our design, by making a specific distinction from what agents actively avoid

dynamically, versus the world that is in their active pool of knowledge, or the constraints in movement caused by the walls so to speak. In a different sense, you could say that people in the real world don't conceptualize every obstacle in their way and plan around them, rather they plot out a path based on the constraints of the walls of a space, and deal with obstacles as they encounter them, which is what the way points variation inherently reflects, but more importantly, creates multiple portal types.

3.2.2.2 Navigation

Similar to Way Portals, portals in our formulation also start with the concept of using line segments to define navigation. Additionally, they both also take into consideration the dynamic circumstances of the situation and thus change from step to step depending on the movements of the agent in each step. More elaborately spoken, depending on the active velocity of the agent and the relative position of the agents to other agents, the preferred velocity of the agent, with respect to the portal, changes from step to step. However, where portals differ from Way Portals is the idea behind the partitioning, and the idea behind the selection of points that are on the portal. In the Way Portal formulation, a Way Portal is conceptualized as a velocity segment and contributes to the potential velocities as part of a linear program for typical velocity obstacle approaches to simulation. On the other hand, in the case of portals for CBRs, the portals are defined to cater to various situations in addition to being a velocity segment.

A good example of this might be the case where a path forks into two paths that would end in the same place with little deviation (see Figure 4). In cases like this, typical

path planning would select one of the orange circles to get to the star. In some simulations you may have congestion and position factored into this, wherein which the agent would stick to either one node, or the other, and then follow it faithfully with some possibility of reevaluation. On the other hand, if we look at these two paths holistically, as if these two nodes were one node instead, there is only one path for the agent to select. By functionally determining which node to travel to (presumably by some distance metric), the path of the agent doesn't need to change at all (at least from a topological perspective). In fact, as forces from other agents move the agent in question, the evaluation of the point on portal selection function would produce differing results defining whether to enter through one side or the other. If we observe the yellow dotted lines in Figure 4, they both denote a partition to the same topological areas between the green region and the purple region, logically, it's not that big of a leap to combine the two portals into one as they essentially represent the same high level mapping.

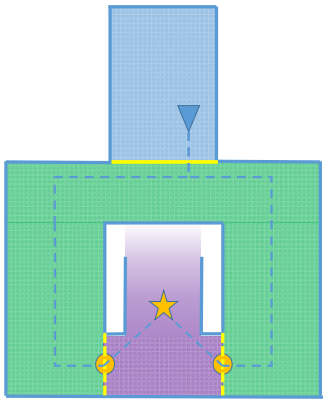

Figure 4: Portal composed of two segments

This type of portal condensing lends itself to less intensive path planning, especially in cases where the planning is more dependent on congestions and other factors than what static metrics would define with regards to the path length itself. Additionally, it also gives a new conceptualization of portals that is slightly different from Way Portals in the sense that our portals may consist of multiple line segments (like a piece wise function) or may have more complex and varied functions behind choosing a destination point on the portal. The examples of these complex functions could be numerous. One example might be a sinusoidal function to define alternating exiting and entering turn stiles. Ultimately, it leads different types of portal objects.

3.2.2.3 Modeling and Path Planning

Because the division of the regions in the environment have to be divided topologically, the actual division of the regions themselves were embedded into the modeling. Intuitively, this partitioning of topological areas also allows for a rough, yet natural division of the geometry. As such, regarding the actual geometry itself, it was not necessary to create an acceleration data structure since the collision geometry could be easily partitioned into their respective topological regions. However, the real challenge was in how to divide these regions appropriately into various regions. In our work, we simplified the geometry and did the partitioning of the regions manually into divisions that we thought were more or less intuitive.

As discussed before, the path planning was done based on the topological regions themselves. At run time, within the mental model of the agent, the path of the modeled

agents was something akin to “First let’s leave our seats, than go down the ramp, to the back hallway, and finally, make our way to the exit on the left.” In that sense, the path planning is very intuitive and on the level of what a person would naturally think when navigating a building.

The roadmap construction itself was done based on the connectivity of the regions and the weight of each portal. To find shortest paths, we hand weighted the portals based on metrics of congestions from observed video data as well as general intuition on general path perception. The smaller details (e.g. Figure 4) were calculated based on the function for particular portals at run time.

Because the number of regions in our models was fewer than what would exist in a typical roadmap, the portals were more or less done by hand, with some automation via material tagging for characterizing different elements of congestion. Between various regions, the path planning was precomputed beforehand. To do this, each portal connecting various regions were given particular weights. If we conceptualize this from the graph theory point of view, portals correspond to edges in the connectivity graph and regions correspond to the nodes in the graph (which we emphasize as also being topological areas in the graph). As in the case in the real world, each agent had knowledge of multiple paths. The multiple paths were calculated using Yen’s K-shortest path algorithm [19]. Since the initial path planning strictly involves only the static knowledge of the agent, these paths were precomputed and placed into the agents as

static knowledge of the world on the simulation start. The more dynamic elements of the path planning take place using agent knowledge and portal functions.

3.3 Holistic System Design

Armed with enough detail about the various relevant components, we can talk about the system itself in more detail. As mentioned before, the Agent object is the base unit of the simulation, which contains the cognitive model described in section 3.1.2.1. Figure 5 shows the overall flow of interaction and components of the various parts of the system at a high level.

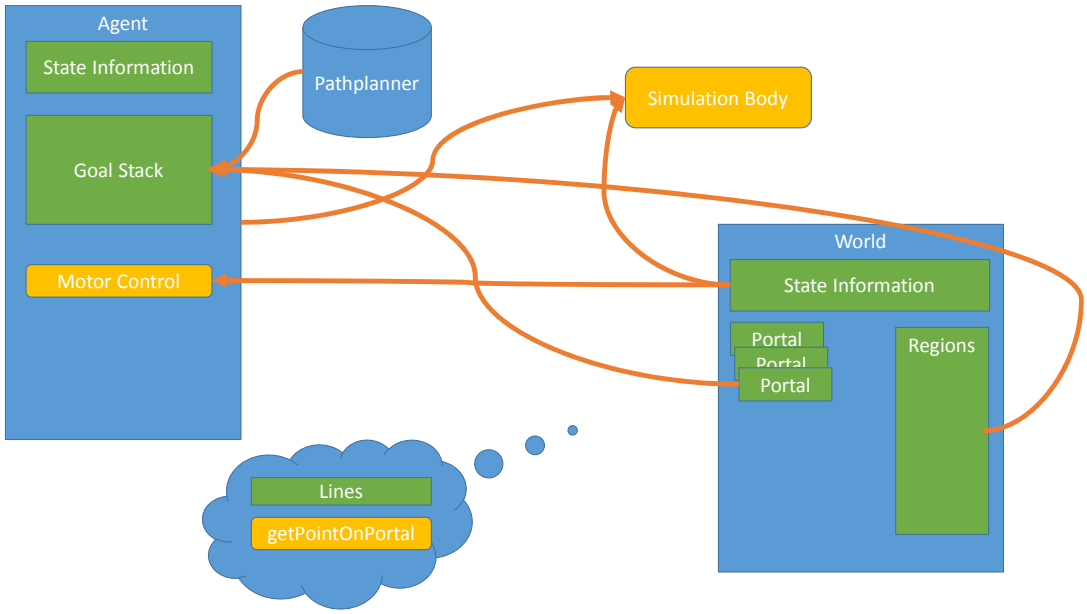

Figure 5 General Program Design

3.3.1 The Path Planner Class

The path planning (saved static paths) are stored in the Path Planner class, which is accessed by the cognitive model, specifically, they are accessed by whatever goals that need path planning information. Similarly, the goals also access the world information necessary for items like selecting relevant portals that are in the world.

3.3.2 The World Class

The world, as we mentioned before, is partitioned into topological regions. However, from the design aspect, the world is more complex. The world itself contains a list of Region objects (which are the topological regions), a list of Portal objects (which are the boundaries that connect various regions together), the current state of the world (namely time), as well as a list of Agent objects that exist in the world.

3.3.2.1 Region Class

The Region class contains the geometry associated with the region as well as references to the portals that separate the region from other regions (indirectly accessed from the world class). This class is mainly used as the building blocks of the world, and are accessed via the world class. The regions themselves are identified via unique identifiers (id numbers) used for efficient access of the components of the region.

3.3.2.2 Portal Class

The portal class contains the data of the various line segments that define their locations in the world, as well as a list of id numbers that identify all the regions they connect. The most important aspect of the Portal class is its "getPointOnPortal" function. This function, depending on the type of portal, will return a point on the portal for an agent to travel to.

3.3.2.3 Agent Class

The agent class was more or less described in detail when describing the cognitive model. But to quickly summarize the components of the Agent class, it consists of its current state (velocity, desired velocity, position, and goal stack). Functionally, it consists of its various behaviors (motor controls), which are invoked by the simulation body and the various goals in its stack. Agents themselves currently detect each other in the most simplistic method possible, which is an n^2 approach of detecting each other neighbor.

3.3.3 Simulation Flow

Now that all the various components have been introduced, the crowd simulation body shown in Figure 1 should make more sense (reprinted here for convenience).

```
01 inline void bodySim(int i){
02     Agent* curAgent = World::getAgent(i);
03     Goal* currentGoal = curAgent->getCurrentGoal();
04 
05     Action* theAct;
06     bool wasExecuted = currentGoal != NULL;
07     //Check if the agent has any goals in memory
08     if(wasExecuted){
09         theAct = currentGoal->getNextAction();
10         //Set up conditions for acting
11         theAct->execute();
12     }
13     //If the goal hasn't been completed
14     if(!curAgent->currentGoalComplete()){
15         //run agent's unique behaviors
16         curAgent->act();
17     }
18     if(curAgent->isMoving()){
19         curAgent->position() = curAgent->position() + (curAgent->vel())*World::getDeltaT();
20     }
21 }
```

Figure 6 Crowd Simulation Body

However, now we can summarize the flow in more detail. Each agent runs on a single thread, which is motivated by a goal, executed per agent. A goal of "Go To Region" with relation to the simulation body works like this:

1. The goal accesses the Path Planner class to retrieve the route the agent will take to its destination region (initial construction of the goal has this imbedded).
2. With a route determined, the goal gets the adjacent regions that the agent is in and from the region retrieves the relevant portal to get to the next region in its path (line 9).
3. It is at this stage where the selected portal is invoked to get the desired point on the portal to travel to (line 9).
4. The goal then generates an action with a desired velocity set to the chosen point on the portal (line 9).

5. When the action is executed (line 11), the agent's state changes, specifically, a change in desired velocity.
6. Finally, when the agent "acts", the agent's state is changed (typically via movement).

From the above system design, there's clear, yet clean object oriented approach to the simulation. However, the big question in this is "how can such an object oriented design smoothly transfer to the GPU?" To address this question, we have to take a slight detour from talking about the crowd simulation to a discussion on OOSoAs.

3.4 Object Oriented Struct of Arrays (OOSoA)

As mentioned before, it is well known that in larger applications where several individual objects can't fit into the cache, a SoAs approach becomes more efficient. This fact is brought to the forefront even more in GPU programming due to coalesced memory access patterns being more efficient on the GPU due to the inherent structure of the GPU [8].

Because GPU program performance is so affected by the memory access patterns, especially to global memory, it makes traditional OOP infeasible. The main unfortunate loss in this situation is the programmability and maintainability of the code. Moreover, agent based simulation lends itself to object oriented programming as the programmer can imagine the agents as individual agents. When agent based models are programmed in other methodologies, the code becomes less intuitive. This of course brings about the

question of whether there's a way to apply the popularized Object Oriented Programming (OOP) paradigm to GPU based programs. The answer that we've come up with is the Object Oriented Struct of Arrays (OOSoAs), which is the crux of transferring our crowd simulation framework to the GPU.

3.4.1 Object Oriented Struct of Arrays Idea

The idea behind OOSoAs is to hide the SoAs structure behind accessor methods. This in part, was inspired by the thread indexing structure used on the GPU. Common convention for GPU programming involving shared memory is that each thread is more or less assigned to write to the equivalent shared memory index as the thread id. We apply a similar concept for OOSoAs. The effect of this becomes having each object have a unique identifier (an integer) through which the object can be accessed. However, this access is done via accessor methods. By doing this, the accessor method hides the identifier, so that when threads "access" the object, on the surface, it appears to be accessing an actual object as opposed to accessing an array, or a struct of arrays.

For the sake of simplicity, we can take a simple vector object as an example of this (see Figure 7). In this example, we simply use references for purposes of length of the code, but it is fairly obvious that you can split them into getters and setters. More importantly, from this example you can see that the SoAs (the "Vectors" struct), is hidden behind the "Vector" class.

Another interesting aspect to note is the effect of nesting OOSoAs together as what would exist in more complex programs. An analysis of the effects on the performance of the program will be covered in Chapter 4.

```
typedef struct Vectors{
    float* x;
    float* y;
    float* z;
}Vectors;

Vectors vectorSet;
__device__ Vectors d_vectorSet;

class Vector{
public:
    size_t id;
    Vector(){
        id = 0;
    }
    Vector(int i){
        id = i;
    }
#   ifdef __CUDA_ARCH__ //For transparency between CPU and GPU code
    __device__ float& x(){
        return d_vectorSet.x[id];
    }
    __device__ float& y(){
        return d_vectorSet.y[id];
    }

    __device__ float& z(){
        return d_vectorSet.z[id];
    }
#   else
    __host__ float& x(){
    return vectorSet.x[id];
    }
    __host__ float& y(){
        return vectorSet.y[id];
    }

    __host__ float& z(){
        return vectorSet.z[id];
    }
#   endif
    //cross this vector with v, storing the result in dest
    __host__ __device__ void crossProduct(Vector v, Vector dest){
        dest.x() = y()*v.z() - z()*v.y();
        dest.y() = z()*v.x() - x()*v.z();
        dest.z() = x()*v.y() - y()*v.x();
    }
    //dot this vector with v, storing the result in dest
    __host__ __device__ float dotProduct(Vector v){
        return x()*v.x() + y()*v.y() + z()*v.z();
    }
};
```

Figure 7 Vector Example of an Object Oriented Struct of Arrays

Chapter 4 Analysis

In this section, we go over an analysis of crowd simulation as a whole, going over design aspects, performance, etc. Specifically, we start with an analysis of OOSoAs and then go into an analysis of the Crowd Simulation itself with regards to both performance and behavior.

4.1 Analysis of Object Oriented Struct of Arrays

In this section we analyze the Object Oriented Struct of Arrays with respect to performance. In this section, we discuss a small case example as a proof of concept. We separate the analysis of the OOSoAs from the crowd simulation to make clear the effects of the OOSoAs.

4.1.1 Object Oriented Struct of Arrays Comparison

To analyze OOSoAs, we produced a simple example of vector dot products using a Struct of Arrays (SoA), an Array of Structs (AoS), and an Object Oriented Struct of Arrays (OOSoA).

4.1.1.1 Code Style Analysis

So the first thing we address is the actual layout the code using each style of coding, as that is one of the more important aspects of OOSoAs vs. SoAs. In this example, the vectors in the first half are being crossed with the vectors in the second half.

First, note the conditional compilation between lines 19 through 41 of Figure 7. By using accessor methods, we can improve the usability of the code, easily alternating between GPU and CPU code. If you note the cross and dot product code, it can be used on both the CPU and the GPU because the accessor methods are the same. This is extremely useful for the purposes of debugging, as you can alternate between CPU and GPU code to identify problems.

From Figure 8, you can see implementations of all three methodologies, SoAs, AoSs, and OOSoAs. The most observable thing is that the SoAs implementation is more cumbersome then the other two, which is expected. On the other hand, the AoSs and the OOSoAs implementation look rather similar in simplicity. If we extrapolate this to more complex code, than the SoAs, as you could imagine the code becomes extremely muddled. So as far as code readability, especially when objects start interacting with each other, and the complexity of such objects increase, AoSs and OOSoAs have the upper hand.

```
//CODE BLOCK A
//cross product using a Struct of Arrays
__global__ void crossProductKernel(size_t numVectors, Vectors vectorList, Vectors
result){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if(idx < numVectors/2){
        result.x[idx] = vectorList.y[idx*2]*vectorList.z[idx*2+1] -
                        vectorList.z[idx*2]*vectorList.y[idx*2+1];
        result.y[idx] = vectorList.z[idx*2]*vectorList.x[idx*2+1] -
                        vectorList.x[idx*2]*vectorList.z[idx*2+1];
        result.z[idx] = vectorList.x[idx*2]*vectorList.y[idx*2+1] -
                        vectorList.y[idx*2]*vectorList.x[idx*2+1];
    }
}

//CODE BLOCK B
//cross product using glm vectors
__global__ void crossProductKernel(size_t numVectors, Agent* vectorList, Agent* result){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if(idx < numVectors/2){
        result[idx].velocity =
                  glm::cross(vectorList[idx*2].velocity,vectorList[idx*2+1].velocity);
    }
}

//CODE BLOCK C
//Implementation of OOSoAs cross product function
__host__ __device__ void crossProduct(Vector v, Vector dest){
    dest.x() = y()*v.z() - z()*v.y();
    dest.y() = z()*v.x() - x()*v.z();
    dest.z() = x()*v.y() - y()*v.x();
}
//cross product using OOSoA cross product
__global__ void crossProductKernel(size_t numVectors, Vector* vectorList, Vector*
results){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if(idx < numVectors/2){
        vectorList[idx*2].crossProduct(vectorList[idx*2+1], results[idx]);
    }
}
```

Figure 8 Code block A shows a typical Struct of Arrays, Code block B shows typical Object Oriented Programming, and Code block C shows OOSoAs.

4.1.1.2 Performance Analysis

However, despite how clean the code may look, performance becomes a large factor, especially when trying to get performance out of the GPU. So how do these implementations compare to each other. So from a theoretic perspective, the reason why AoSs is slow on the GPU is because as the size of the struct increases in size, the stride to access each element in the structure increase [8]. Since the stride increases, the number of global memory access per element access increases. A SoAs avoids this by minimizing the size of the struct such that access to all of a particular field of an "object" are put together. In our simple case, accessing all of x, then all of y, then all of z. As such, the memory access stride doesn't increase as the "object" size increases (although again, it's not actually an object). With an OOSoAs, this same statement can be said to be true. However, there's another factor at work at this point. To mask the SoAs that the OOSoAs is hiding, we need to access the unique identifier, which is an extra global memory access (albeit, still a coalesced memory access). This has two effects, the first, is that at a minimum, the transaction for each access will take twice the amount of time as with a SoAs. The second effect is that the identifiers for each object will take up space in the cache (assuming that we don't cache the values in code). In a worst case scenario, the identifiers take up the entirety of the L1 and L2 cache, and you always have go back to global memory. However, at the same time, the memory accesses have the potential to be masked by increasing the occupancy.

Another issue is when objects are fields of other objects. In the case of AoSs, this isn't an actual issue due to the stride size. In the case of a SoAs, this becomes messy if the "objects" fields have different indices from the thread, or can't be statically determined, than the access would start being very similar to that of OOSoAs. On the other hand, the object nesting uses the same index (e.g. the thread index), then the access time doesn't increase. OOSoAs on the other case, will always act under the former case for SoAs, that is, you are forced to chase down indices. In this case, for each depth of objects being fields of other objects, you double the access time (based on chasing down pointers). Generally speaking though, if the complexity surpasses one or two levels of depth, there is either something wrong with the design of the program, or the program is becoming so complex that the main cost isn't calculations, but branching and memory access.

With that analysis in place, we can look at the analysis of the vector cross product case and see the theoretical analysis as it applies to the performance analysis itself. In our work, we are primarily testing performance with regards to the following:

- Increase in number of fields
- Increase in size of fields

Increasing the size of the fields was done by creating a struct that consisted of an arbitrarily large array (while access itself was still only to the first three vector elements, x, y, and z). Increasing the number of fields was done in a similar manner, except we added extra fields. In the case of either situation, the results should more or less be the

same. However, before we go into this analysis, we need to define a base line of comparison which we establish below. These experiments were conducted using the Nvidia GTX 765M (Kepler architecture).

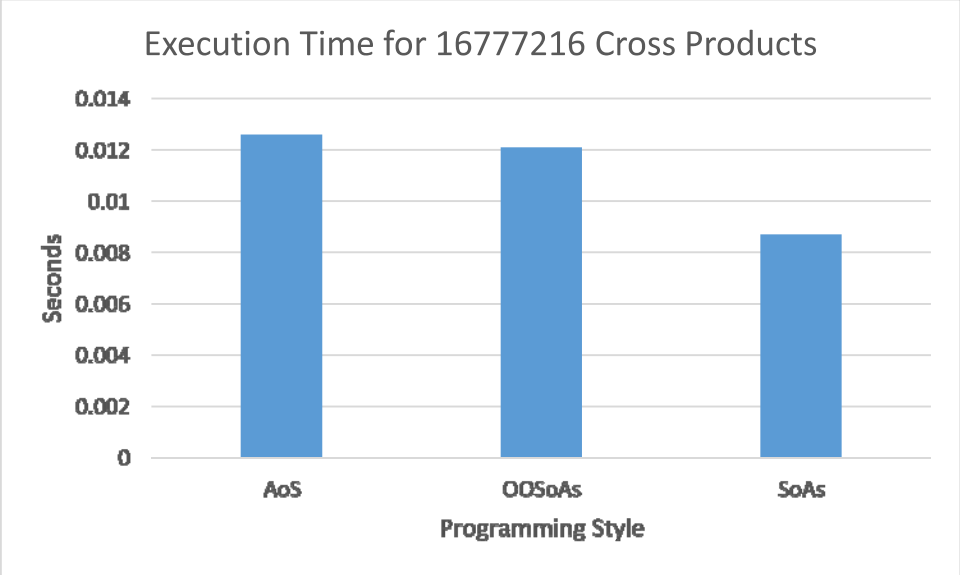

Figure 9 Graph shows the comparative performance of AoS, OOSoAs, and SoAs

Figure 9 shows the relative performance of the three programming styles. The choice of 16,777,216 cross products was to make sure that there was no impact from the L1 cache performance. The actual cache performance on level 2 for each of these programming styles were roughly equal to each other, 65% hit rate with a standard deviation of 1%, so roughly similar. From this graph, it's observable that OOSoAs and AoSs have around the same performance and the SoAs has the best performance (a roughly 1.4x speed up). To best see how this works, the best things to look at would be memory request replays and the instruction statistics.

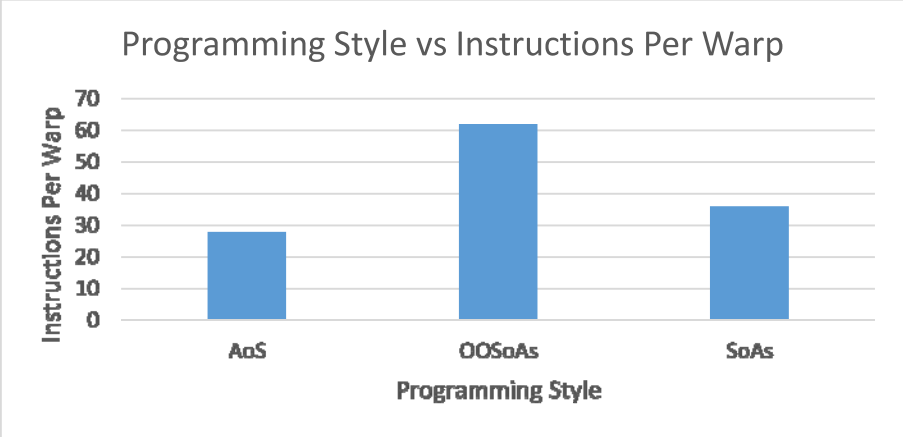

Figure 10 Programming Style vs. Instructions Per Warp

As you can observe from Figure 10, compared to the AoS and the SoAs, the OOSoAs does roughly double the amount of instructions. If we recall the analysis we did prior, this makes sense as there needs to be twice the number of memory requests to retrieve the same amount of data (one memory request for the identifier and one for the actual data). So from this perspective, it make sense that the OOSoAs takes more time. However, given over double the number of instructions as AoS, OOSoAs still manages to keep up with respect to performance. This is can be explained by looking at the number of memory replays that are needed (shown in Figure 11). The first three sets of bars represent the actual numbers as given by Nsight. However, as we said before, OOSoAs inherently uses double the number of transactions (assuming object depth of 1). However, even given the theoretical number of transactions, the number of transactions, that OOSoAs uses is roughly only half that of an AoSs. With double the work, and half the number of memory transactions, OOSoAs and AoSs come out to be about the same in this particular case.

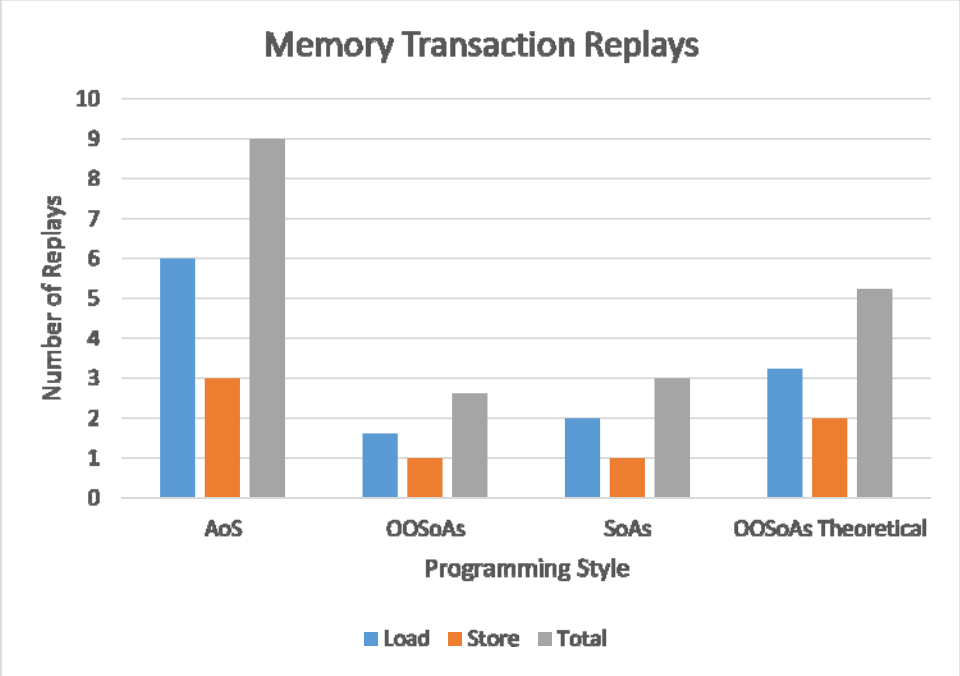

Figure 11 Memory Transaction Replays

From the above figures, we can see that overall, OOSoAs in the simplest case lies at around the same performance as AoSs, with minimal improvement. However, where OOSoAs begins to distinguish itself from AoSs is when the number of fields in the class increases, which is what we'll look at next.

So first off, the number of fields, when increased within reason, had no different effect than increasing the actual size of the struct after our testing. Increases in the struct size itself, affected neither the SoAs nor the OOSoAs programming style, rather, the performance stayed the same. So, we simply increased the size of the struct (in increments of 32 bits) from the original base size of 96 bits for the purposes of analysis.

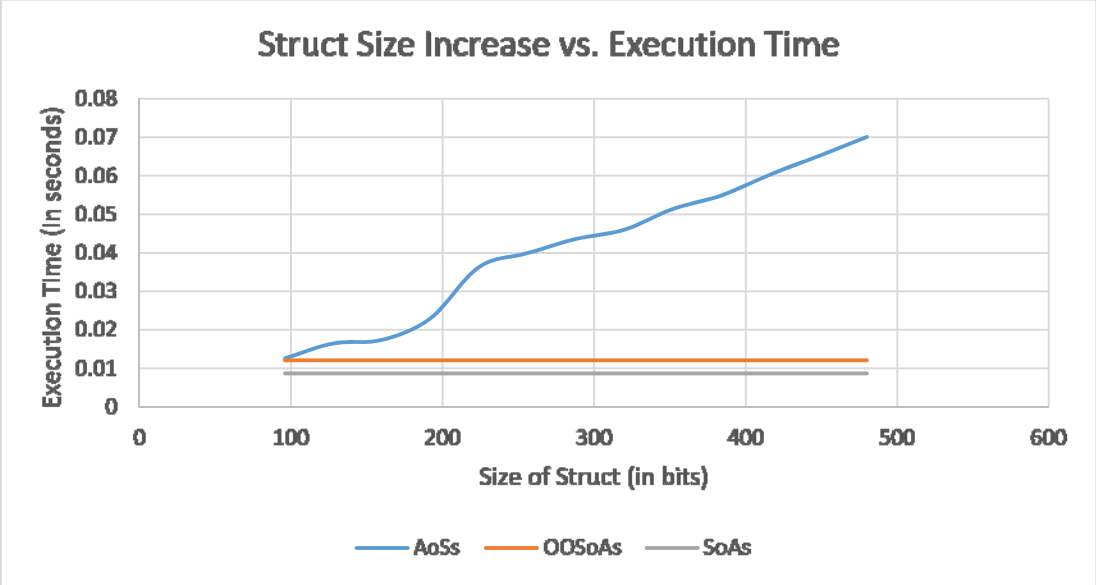

Figure 12 Struct Size vs Execution Time

Taking a look at the actual numbers, you can see that the OOSoAs and the SoAs maintain their performance regardless of the struct size. On the other hand, the AoSs increases significantly in size. This is where OOSoAs takes a significant advantage over AoSs. Additionally, although comparatively slower to the SoAs approach (although significantly faster than an AoSs approach), the code becomes more maintainable and easier to read. So as a compromise between the programmability of the AoSs approach and the performance of the SoAs approach, we reach a strong middle ground that allows for stronger programmability than a SoAs approach while capitalizing on many of the advantages of Object Oriented Programming.

Another interesting result to look at for the analysis are the stall reasons for the three different design patterns (shown in Figure 13), which further emphasize the main points we are asserting. From here, you can see that the memory access is the largest

bounding factor for the performance in any case (combinations of Memory Dependency and Memory Throttle), in all three forms. Particularly from the SoAs pattern, we can see that the memory limitations in performance are more or less unavoidable at this point. Observing the occupancy, we get occupancy above 86%, suggesting that there is not much that can be done with the throughput either.

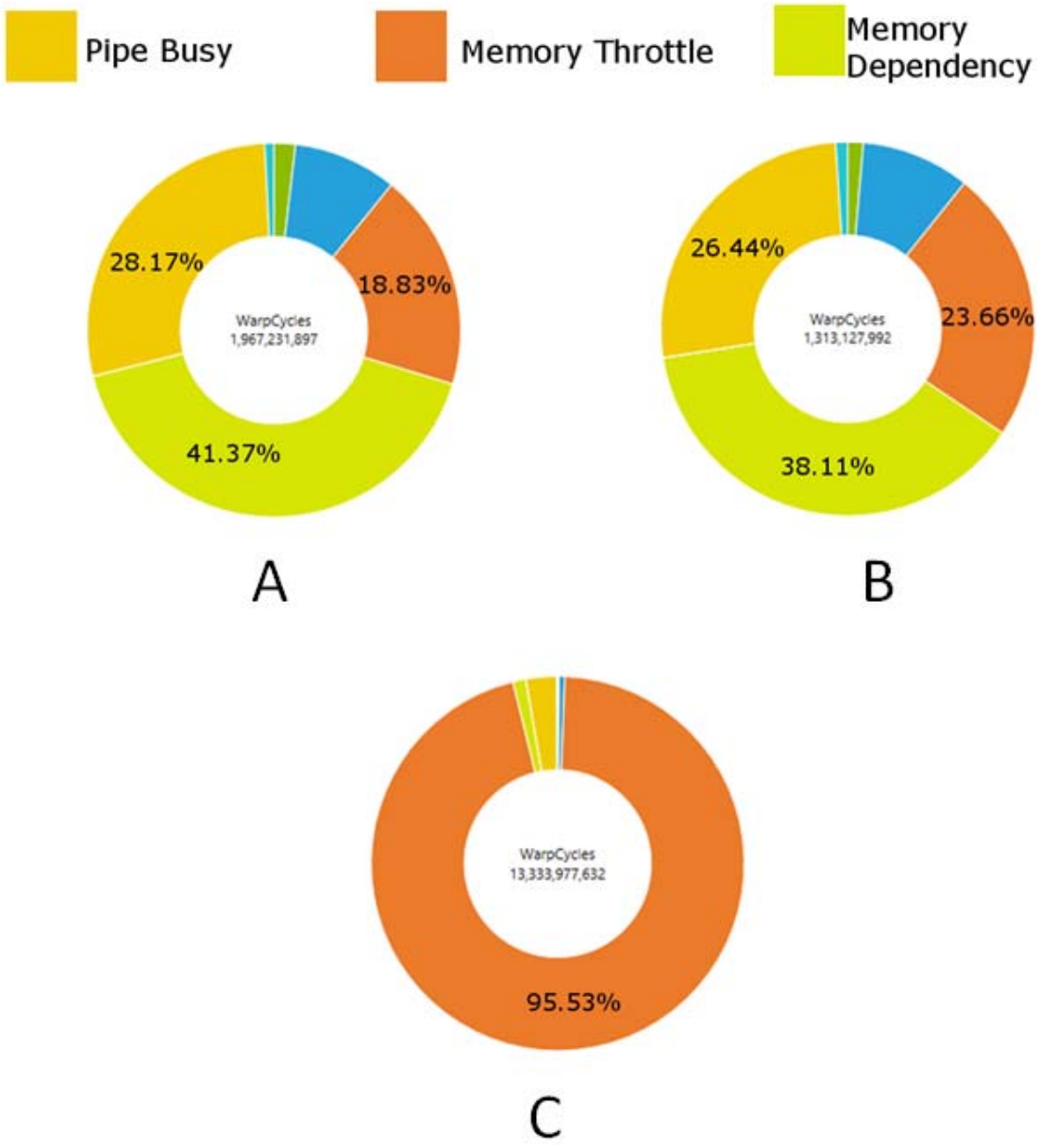

Figure 13: a) OOSoAs, b) SoA, c) AoS

So in this section of the analysis, we showed that the OOSoAs design pattern keeps most of the advantages of the SoAs approach while maintaining programmability of the system itself. So, using OOSoAs as the building blocks of our code, we can move

the heavily Object Oriented Design that we describe in Chapter 3 onto the GPU with relatively little performance hit, and a significant increase in programmability.

4.2 Crowd Simulation Performance Analysis

In this section we talk about the GPU performance of the Crowd Simulation relative to the CPU, and the speed-up process.

4.2.1 Pure Object Oriented Crowd Simulation

In our initial design, we built the program on the CPU using traditional OOP, the design that we describe in Chapter 3, with rough parallelism using OpenMP.

The performance of this program was mediocre, but it gives us a starting point. Roughly, it was about 25fps for 2000 agents. The next step we took was to move this infrastructure directly to the GPU, with as little change as possible, to test how the program ran on the GPU. Given the complexity of the design with respect to object oriented programming, we didn't expect much speed up. However, the results exceeded our expectations in that the program ran significantly slower than it did on the CPU, the order of 0.5 FPS for roughly 2,000 agents. However, considering the memory access patterns, this should have been expected. Although NVIDIA incorporated OOP on the GPU, there were little, if any, optimizations to make OOP work well on the GPU beyond the superficial support for polymorphism. We don't do an in-depth performance analysis of this because the main bottle neck is memory-dependency which has an obvious

optimization of coalescing the memory access. Instead, the next step to look at is how the optimization of OOSoAs changed the performance of the program.

4.2.2 Object Oriented Struct of Arrays Crowd Simulation

So in this section, there are two major points. The first major point is the raw comparison of the GPU and the CPU performance of the Crowd Simulation, to see the speedup. For the sake of consistency, the implementations of the simulation on both platforms were done using the same OOSoAs architecture. In actuality, aside from the initialization code lines necessary for the GPU memory (using new vs cudamalloc), the CPU and GPU simulation code was identical. The second major point, probably the more interesting point, is analyzing the performance limits on the GPU by doing an in-depth performance analysis of the simulation's performance on the GPU. The results for the GPU were gotten from an NVIDIA GTX Titan card. The CPU results were gotten from an Intel i7-3820 3.6-3.8GHz with 10Mb cache and 32 GB RAM.

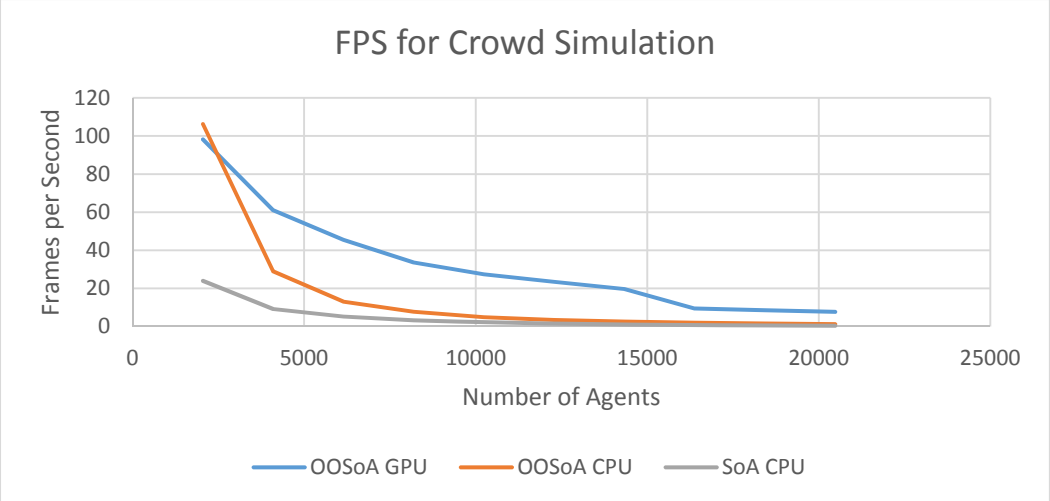

Figure 14: FPS of Crowd Simulation

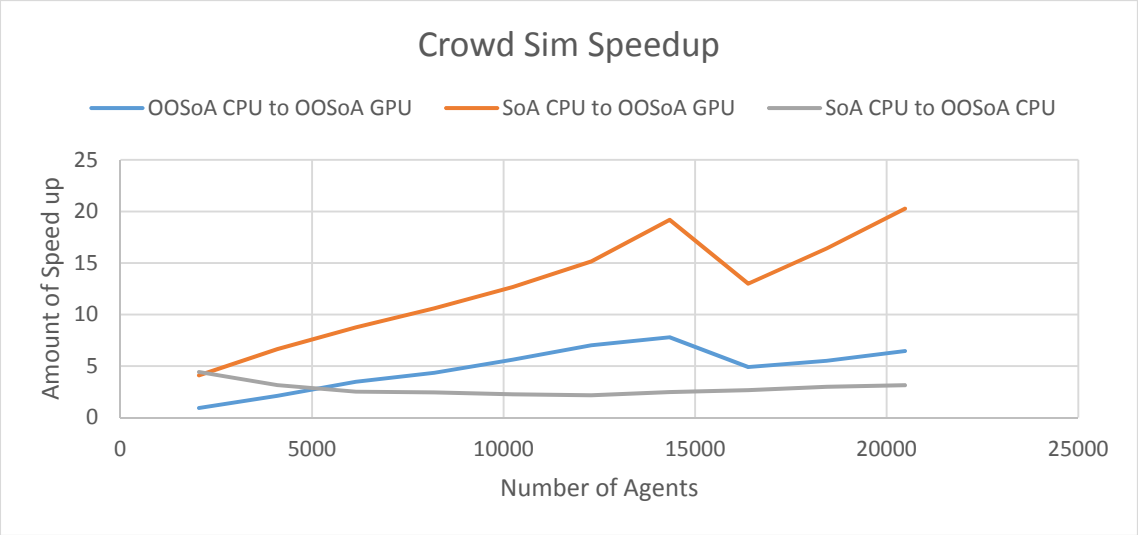

Figure 15: Speedup of Crowd Simulation

Because we were using a new design pattern, we had to build the program from the ground up again, this time using OOSoAs as opposed to a SoAs approach. The results were promising as there was a drastic increase in the performance, both on the CPU and the GPU as what can be seen from Figure 15. For ease of observation for OOSoA CPU to OOSoAs GPU, we've included a separate figure.

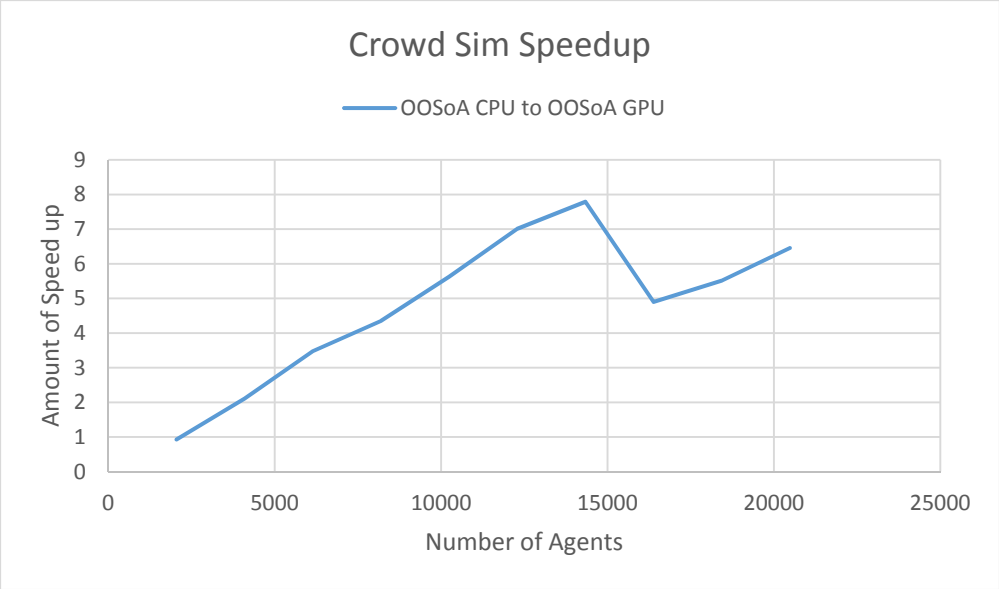

Figure 16: OOSoA CPU to GPU speedup

From Figure 16, you can clearly see the linear speedup that you can achieve from the CPU to the GPU using the OOSoAs design pattern. Note the dip in results around the 12k mark for the number of agents. This represents the point where the cache needs to swap out a large portion of memory. However, the speed increase still maintains a linear trend.

The more interesting facet of the OOSoAs methodology is analyzing the overall performance, and seeing what questions this brings up. Using OOSoAs, we can port agent based simulations to the GPU more easily than before. Because of this, we are enabled to take a closer look at how an agent based simulation itself performs on the GPU without concerning ourselves as much with the actual architecture and its effects on the performance.

The first default thing to check regarding performance are the memory access patterns. From the Nsight profiler, the transaction requests were 1 per load and 1.62 per store (as opposed to the worst case of 32 for each). This means that overall, the transactions for memory were relatively good, even if we double the number of transactions to account for the indexing overhead from OOSoAs. So at the very least, the memory access patterns were good.

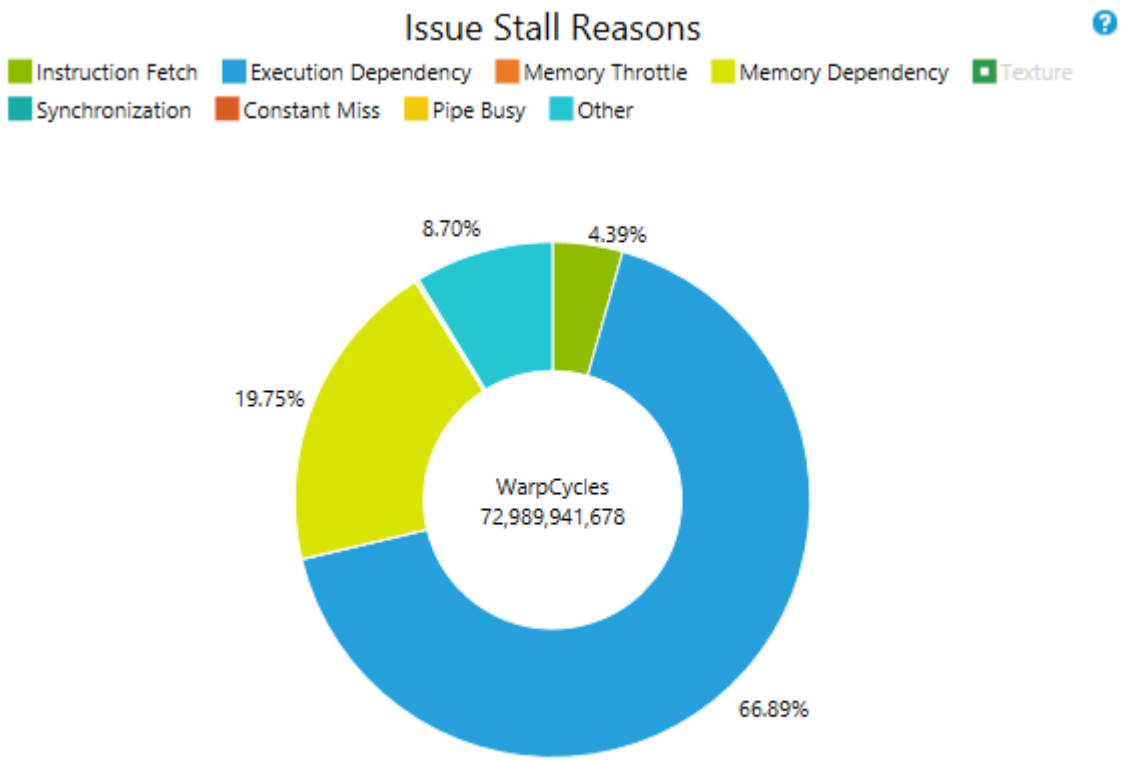

Figure 17 Stall reasons for the Crowd Simulation

Looking at the cache hit rates for our program, both L1 and L2 cache performance were above 97%, meaning extremely good cache performance. The last memory access item to check are the stalls caused by memory access.

Figure 17 is a direct copy of the output of stall reasons as revealed by Nsight. The first thing to notice is that memory dependency is not the largest stall issue in this program (although, it is the second largest stall issue). However, if we recall from Figure 13 from the cross product test, the memory dependency is likely unavoidable. Additionally, the occupancy of the application is around 50%, which should in theory, hide most memory latency. Instead, the most interesting stall reason to look at is the execution dependency stall reason. An execution dependency is a stall reason caused by lack of instruction level parallelism, that is, the next instruction is dependent on the output of the previous instruction. This indicates that the issues lie somewhere outside of the actual memory dependency issues. Also, if we look at the pipe utilization (shown in Figure 18), we can see that roughly only $\frac{1}{3}$ of the pipe is actually being used. Moreover, the majority of the pipe that is being used is being used for arithmetic operations. This tells us two things. The first is that the memory isn't actually the bounding factor. More likely, because of the lack of instruction level parallelism, the memory latency isn't getting hidden by actual work, because not enough arithmetic instructions are happening. Support for this train of thought also comes from the IPC statistics of the crowd simulation. Note that instructions are getting executed almost as soon as they are placed on the GPU, as approximately 1.37 instructions are issued per cycle and 1.3 of them are executed per

cycle. Since the GTX Titan card has four warp schedulers, this indicates that the warp scheduler also isn't being fully utilized, thus pointing to problems with the independence of the arithmetic instructions.

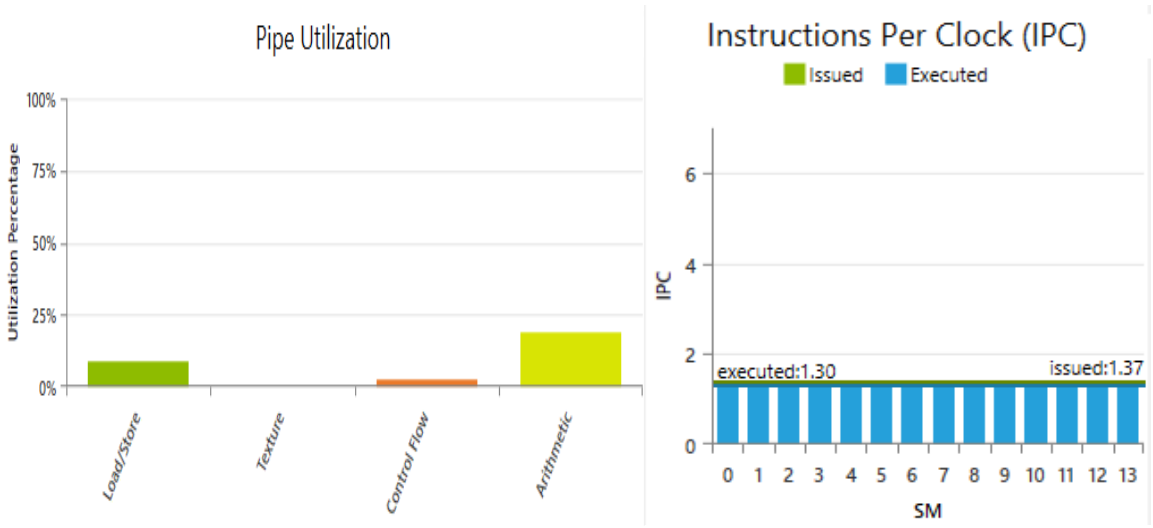

Figure 18 Crowd Sim. Pipe Utilization and Instructions Per Clock

The analysis above all points to the main problem being the instruction level parallelism achievable in the code. The question then becomes, why is there a low instruction level parallelism and how can this be fixed. In general, there are two likely reason for why there isn't a high level of instruction level parallelism. The first is that the instructions can't be parallelized because the dependencies are necessary. The second probable reason is that there aren't enough operations in the first place to capitalize on the entirety of the GPU. In this context however, we can rule out the second reason, because it is readily known that crowd simulation takes a lot of computational power, especially when using agent based methods. In this section, we deduce that the speed

issue for agent based simulation is the actual underlying methodology used to calculate the values necessary for the simulation.

Chapter 5 Conclusion

In the analysis section, we analyzed the various aspects of an GPU based multi-agent large scale crowd simulation using an OOSoAs software design approach and curve based roadmaps as the underlying navigation structure on top of Autonomous Pedestrians framework. In this chapter, we go over the conclusions we can draw from our analysis, summarize our research contributions, and list out directions for future work.

5.1 OOSoAs

5.1.1 Conclusions

From our analysis on OOSoAs, we saw that even in the most basic scenario, they break even with AoSs. With further investigation, we found that while not as fast as a SoAs approach (which was a given), OOSoAs scale just as well as a SoAs approach. However, an OOSoAs approach scales as well as a SoAs approach (the rate of change to data set size stays the same).

5.1.2 Future Work

One of the aspects of OOSoAs that we did not explore was the calculation on the effects of increased object depth on the performance of OOSoAs. This could prove to be

interesting to asses with regards to how the complexity of the objects affect the actual performance of the memory access patterns as well.

Another interesting point of OOSoAs that could be approached is how well polymorphism works in the context of OOSoAs. Since OOSoAs covers the problems regarding the actual memory access patterns of using a SoAs, it leaves us able to explore the polymorphic aspects of Object Oriented Programming on the GPU and see whether OOSoAs will truly allow relatively good performance on the GPU while still supporting a free no-holds no-bars pure object oriented framework.

Additionally, our showing the feasibility of applying the OOSoAs design pattern to a complex crowd simulation also reveals an interesting facet of the OOSoAs beyond just the analysis and speed up of a program on the GPU. OOSoAs was able to maintain the design integrity of the crowd simulation while still achieving speed up, letting us cut to the core of the actual problems surrounding agent based simulation on the GPU. However, this shows interesting factors when regarded to how the design pattern is applied. The core motivating factor behind OOSoAs is to facilitate programming on the GPU. Additionally, this means that we should be able to retrofit objects in object oriented code with an OOSoAs code. In our crowd simulation, this elicits a linear speed up ranging from 5 to 10 times that of the CPU. In other code that is parallelizable, this could possibly generate even greater parallelization. Moreover, there's the possibility of automating the application of the OOSoAs design pattern to other, parallel CPU code, possibly as an option in between typical GPU code and OpenACC.

5.2 Conclusions on Agent Based Simulation Using OOSoAs

5.2.1 Conclusions

In our discussion of GPU based multi-agent simulation using the OOSoAs design pattern, we show that it is possible to attain significant speed up from the CPU to the GPU while maintaining a relatively complex object oriented approach to the design, making the code both more readable and more maintainable. Moreover, we do an in-depth performance analysis. The analysis shows that OOSoAs doesn't actually get in the way of performance analysis, in fact, it actually simplifies the analysis by not only ruling out common memory access problems on the GPU, but also by simplifying the appearance and readability of the code, allowing someone to delve down a lot deeper into the analysis of the code itself to find the real issues in the code.

5.2.2 Future Work

However, in our analysis of the agent based simulation that we implemented using OOSoAs, we found that the major limiting factor of the performance was not the amount of computation, but the amount of instruction level parallelism. The most readily obvious method to use is to just reduce the amount of computation necessary [10] [4] [5] [3] [2] [1], which in turn reduces the problem in size. However, in our work, we're not interested in reducing the problem size as it has been done many times before; we are more interested in why our agent based method can't leverage the more of the GPU's

power and what measures we can take to resolve it, while maintaining an agent based approach that is cleanly designed from an object oriented perspective. This introduces a future question that this brings up from our work; can we leverage more of the GPU's power while maintaining a cleanly design object oriented agent based simulation.

As mentioned before, the simulation system takes a summative approach on a per agent basis. Similar to a social forces model, the simulation takes the sum of the forces, or, in the case of the more conditional items of Wei's simulation methods, each agent needs to be compared to each other agent (recall that we are still using an n^2 approach for more simplicity in analysis) for purposes of determining conditions such as the most imminent collision. Classically speaking, we can view this as a reduction (summation) problem as far as our approach towards optimizing the simulation on the GPU is concerned. However, the question becomes whether we can find strong design patterns to keep the intuitiveness of our current simulation build, which is one of many possible optimization strategies that would be an interesting continuation of our work.

5.3 Conclusions on Way Portals

5.3.1 Conclusions

With regards to our variation on Way Portals, we concluded two things. The first is that to create a more stable formulation for, we need a better way to partition regions, and create different formulations for the different types of portals before we put it into

practice. The other thing we've learned is that the impact on the simulation itself is relatively small.

5.4 Intellectual Merits and Broader Impacts

Because GPGPU programming is done primarily for large performance gains, the engineering aspects of the GPU programming are largely overlooked. This in-turn leads to code that is both hard to reuse and hard to maintain. OOSoAs helps to alleviate this issue by improving the maintainability of code with relatively little effect on performance, especially when the complexity of the code is more easily written complex objects of considerable size.

Beyond improving the maintainability of code on the GPU, OOSoAs also helps in making GPU programming more accessible. For basic logic bugs in code, the usefulness of alternating between CPU and GPU code is significant, as you don't have to deal with debugging on the GPU if it isn't necessary. Additionally, it allows the application of Object Oriented Design patterns onto GPU code. Even more so, the easy toggling with conditional compilation (possible because of the OOSoAs design patterns), allowing for quick comparisons of the GPU code and the CPU code to run comparisons between CPU code and GPU code.

Additionally, we show that OOSoAs has little impact on the actual performance analysis of the system while simultaneously reducing GPU programming itself into a more manageable form. By making GPU programming more accessible to others, GPU

programming itself can reach a larger audience base. This in turn allows for even more people to approach GPU programming, finding more new and novel applications of GPU programming.

In our current era, object oriented programming is one of the more prolific programming paradigms used. OOSoAs opens the door of possibilities ranging from automated conversion of object oriented designs to OOSoAs that run well on the GPU to broadening the scope of GPU programming to include programmers that don't specialize in GPU programming. By opening the door to a new, larger audience of programmers, we will see broader impacts of not only the proliferation of GPU programming in the world, but also of new uses and interesting problems that get introduced to the GPU that didn't seem plausible to GPU programming before, making for greater heights in the exploration of GPU programming as a whole.

Bibliography

[1] A. Bleiweiss, "Multi Agent Navigation on the GPU," in *GDC09 Game Developers Conference*, 2009.

[2] G. Caggianese and U. Erra, "Parallel Hierarchical A* for Multi Agent-Based Simulation on the GPU.," in *Euro-Par 2013: Parallel Processing Workshops*, 2014.

[3] A. Demeulemeester and e. al, "Hybrid path planning for massive crowd simulation on the gpu," in *Motion in Games*, 2011.

[4] E. Passos, "Supermassive crowd simulation on GPU based on emergent behavior," in *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 2008.

[5] J. Shopf, J. Barczak, C. Oat and N. Tatarchuk, "March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatres on GPU," in *ACM SIGGRAPH Games*, 2008.

[6] W. Shao and D. Terzopoulos, "Autonomous pedestrians," *Graphical models,* vol. 69, pp. 246-274, 2007.

[7] S. Curtis, J. Snape and D. Manocha, "Way Portals: Efficient Multi-Agent Navigation with Line-Segment Goals," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012.

[8] B. Oster, "Advance Cuda Training Nvision," [Online]. Available: http://www.nvidia.com/content/cudazone/download/advanced_cuda_training_nvision08.pdf. [Accessed 15 April 2015].

[9] "OOCUDA".

[10] P. Du, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing,* vol. 38, no. 8, pp. 391-407, 2012.

[11] C. Reynolds, "Flocks, herds and schools: A distributed," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987.

[12] A. Treuille, S. Cooper and Z. Popović, "Continuum crowds," *ACM Transactions on Graphics,* vol. 25, no. 3, pp. 1160-1168, 2006.

[13] W. Jeong and R. Whitaker, "A Fast Eikonal Equation Solver for Parallel Systems," in *SIAM conference on Computational Science and Engineering 2007, Technical Sketches*, 2007.

[14] J. van den Berg and e. al, "Interactive Navigation of Multiple Agents in Crowded Environments," in *Proceedings of the 2008 Symposium on interactive 3D Graphics and Games*, Redwood City, California, 2008.

[15] S. Guy, "Clearpath: highly parallel collision avoidance for multi-agent simulation," in *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, New Orleans, 2009.

[16] W. Shao and D. Terzopoulos, "Autonomous pedestrians," in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2005.

[17] J. Van den Berg, M. Lin and D. Manocha, "Reciprocal velocity obstacles for real-time multi-agent navigation," in *Robotics and Automation, ICRA 2008*, 2008.

[18] J. Van Den Berg, S. Guy, M. Lin and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics research*, Springer Berlin Heidelberg, 2011.

[19] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science,* vol. 17, no. 11, pp. 712-716, 1971.

[20] S. I. Park, Y. Cao and F. Quek, "Large Scale Crowd Simulation Using A Hybrid Agent Model," in *Motion in Games*, 2011.