GPU Accelerated Nature Inspired Methods for Modelling Large Scale Bi-Directional Pedestrian Movement

by

Sankha Baran Dutta

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

Electrical & Computer Engineering
University of Manitoba
Winnipeg, Manitoba

Acknowledgments

I begin by thanking both of my advisors Prof. Robert D. McLeod and Prof. Marcia Friesen. Throughout my research tenure they have been a pillar of support helping me out with valuable suggestions wherever I needed them. They also have the innate ability to motivate and support me without ever pressurizing and by firmly emphasizing quality over quantity. My research tenure under their guidance has been some of the best times of my life as a student and I can only thank them for this enriching experience.

I would also like to thank my other committee members Dr. Ken Ferens and Dr. Rasit Eskicioglu for kindly agreeing to be a part of my defence committee and for making vital suggestions without which this dissertation would never have been complete.

I would like thank the professors whose courses I have taken and their sound teaching helped me to increase my domain of knowledge.

Bandhuli Sen has been my strength even in the most tumultuous of times and I thank her for her presence through thick and thin.

This dissertation would never have been possible without the support of my parents Swadesh Kumar and Sumita Dutta. In my early age they inculcated me the desire to succeed while never compromising on honesty and integrity. I would also like to thank my brother Subhabrata Dutta and my sister-in-law Lipika Dutta for their support. I cannot forget the charming presence of my nephew Saptak Dutta.

I cannot forget the moral support that I get from my childhood friends Sayan Dasgupta, Aindrik Dutta, Avik Samanta, Dipanjan Bhadra, Abhik Mitra, Soumyadeep Ghosh, Suman Roy and Arjun Gupta. I would like to thank for their valuable advice which helped me to make important decisions in my life.

I am also very much grateful to all my laboratory colleagues for their support and I want to thank Shachi Singh, Ryan Neighbour, Hamid Reza Nasrinpour and my other colleagues.

Abstract

Pedestrian movement, although ubiquitous and well-studied, is still not that well understood due to the complicating nature of the embedded social dynamics. Interest among researchers in simulating the nature of pedestrian movement and interactions has grown significantly in part due to increased computational and visualization capabilities afforded by high power computing. Different approaches have been adopted to simulate pedestrian movement under various circumstances and interactions. In the present work, bi-directional crowd movement is simulated where equal numbers of individuals try to reach the opposite sides of an environment. Two pedestrian movement modeling methods are considered. The reasonableness of these two models in producing better results is compared without increasing the computational complexity. First a Least Effort Model (LEM) is investigated, where agents try to take an optimal path with minimal changes from their intended path as possible. Following this, a modified form of Ant Colony Optimization (ACO) is developed, where individuals are guided by a goal of reaching the other side in a least effort mode as well as being influenced by a pheromone trail left by predecessors. The objective is to increase agent interaction, thereby more closely reflecting a real world scenario. The methodology utilizes Graphics Processing Units (GPUs) for general purpose computing using the CUDA platform. Because of the inherent parallel properties associated with pedestrian movement, such as similar interactions of indi-

viduals on a 2D grid, GPUs are a well suited computing platform. The main feature of the implementation undertaken here is the data driven parallelism model. The data driven implementation leads to a speedup up to 18x compared to its sequential counterpart running on a single threaded CPU. The number of pedestrians considered in the model ranged from 2K to 100K, representing numbers typical of mass gathering events. A detailed analysis is also provided on the throughput of pedestrians across the environment. Compared to LEM model, there is an overall increment of 39.6% in the throughput of agents using the ACO model with a marginal increment of 11% in the computational time. A detailed discussion addresses implementation challenges faced and avoided.

Contents

List of Tables

List of Figures

CHAPTER 1

1.1 Introduction

Pedestrian movement and navigating through crowds or parts of crowds seems to be an unavoidable part of modern life. As a consequence, crowd simulation has attracted considerable interest and research. Pedestrian flow related issues arise in different instances, such as in mass-gatherings, sporting events and even cross-walks within large urban centers. As the density of the crowd increases, the vulnerability towards an adverse panic event or disaster also increases. Replicating the situation in the real world to create a safer environment is not always practical. As such, simulation has become an attractive alternative for simulating safety policy and crisis situations when crowds are very large.

However, simulating life-like crowd dynamics is an arduous task. In addition to these real pedestrian movement concerns, producing a realistic pedestrian movement simulation for different scenarios is also gaining importance in the gaming and graphics industries. While modelling the movement of pedestrians, there are several important things to be kept in mind. Pedestrians typically have a goal in their mind towards which they move, but at the same time, they try to avoid collisions or contact with other pedestrians. There are several models that have been proposed to emulate crowd movement including social

force models [1]-[3], cellular automata models [4]-[7], gas kinetic models [9] and agent based models [10]-[14] to name a few.

In this work, a situation was emulated where two groups of pedestrians are placed at opposite sides of an environment. Their goal is to reach the other side of the environment in a fixed number of time steps. To model this bi-directional pedestrian movement situation, two algorithms are used. The first algorithm that is used is known as Least Effort Model (LEM) [15]. LEM is inspired from real world pedestrian movement. It is a simple algorithm where the pedestrians move in the environment with as little deviation as possible from an intended path. This allows the pedestrian to move towards their goal along the shortest path possible requiring the least effort to achieve their target. Subsequently, a variant of Ant Colony Optimization (ACO) [16]-[18] is also used to modify an agent's or pedestrian's strategies by which their objective is accomplished. ACO is a population-based metaheuristic algorithm. ACO utilizes the foraging behavior of ants to find the shortest path between the food sources and their nest. In the natural world, ants communicate indirectly through a process called *stigmergy*, where they deposit a chemical substance (*pheromone*) to construct the shortest path to a food source. Pheromones evaporate after a point of time if not updated by further deposition. So as more ants traverse a particular trail, that trail become more prominent than others and eventually only the shortest trails between food and nest prevails. Through this process, ants develop an optimized path between the food source and their nest.

After its initial development, ACO was first used to provide solutions to the travelling salesman problem [19] and has since then been adapted to several applications areas, such as vehicle routing [20], circuit design [21] and recently in complex applications like

protein folding [22]. The pedestrian movement can also be modelled as ants, considering each pedestrian as an ant or agent that deposits a pheromone trail in the sense of signaling to following agents that the chosen path may be desirable. A comparison between the results obtained using the LEM and ACO algorithms are also provided in the thesis.

Modelling of the pedestrian movement is a resource intensive job. Execution time on sequential machines (single core CPU) is very high. In the bi-directional pedestrian movement application developed here, each pedestrian can make autonomous decisions throughout the simulation. This autonomous property suits the parallel implementation of the algorithms and makes Graphics Processing Units (GPUs) more suitable as the computation platform.

GPUs were initially used for graphics purposes. With the introduction of Compute Unified Device Architecture (CUDA) [23]-[25] by NVIDIA [45], GPUs started to be used for more general purpose computing. Nowadays, GPUs are used for numerous applications where there is ample opportunity to exploit fine grained parallelism. A GPU works as a co-processor to a CPU and can expedite many compute resource heavy processes. GPUs are multi-core processors which is the primary reason behind decreasing the run time of an application. Yet, the presence of GPU based multi-core processors doesn't make the implementation easy or straightforward. GPUs are better suited for data-driven parallelism, where identical threads operate on a data-set. There are several factors that should be taken into account while developing an application on a GPU. These include avoiding warp divergences, avoiding serialization and increasing parallelism by using index operations and other advanced techniques while attempting to maintain 100% occu-

pancy of the GPU. In the following sub-section a synopsis of the work undertaken is provided.

1.2 Thesis Summary

In Chapter 2, the LEM and ACO models are discussed in detail. The objective of this work was to model bi-directional pedestrian flow using LEM and ACO algorithms on a GPU platform in the most efficient way in terms of performance. Subsequently, a comparison was performed between the models running on a CPU and the GPU and the outcomes validated relative to one another, comparing both agent throughput (LEM vs. ACO) and platform efficiency (CPU vs. GPU). Modifications performed to implement the algorithms on a GPU are discussed. Chapter 3 describes in detail how these two algorithms achieve their goal. The NVIDIA GPU GTX 560ti which is based on FERMI [45] architecture is used for the work. In Chapter 4, a detailed description of the processor architecture and how programming methods affect the performance of the GPU is discussed. This helps to understand the implementation discussion of the algorithms in the later chapters. In Chapter 5, a detailed description of the GPU implementation of the LEM and ACO based algorithms to simulate bi-directional pedestrian movement is provided. The task driven and the data driven methods used for the implementation and the optimization techniques adopted for the implementation are described in detail in the Chapter 5. Chapter 6 provides a detailed account of the performance. The GPU execution time for the LEM and ACO is compared while increasing the number of agents in the environment. The performance gain of the GPU over a CPU is also provided in this chapter. Chapter 7 provides results, comparing the output obtained from the LEM and ACO based

simulations. The main result is obtained by comparing the throughput of agents from different simulations, measured as the number of agents that are able to cross to the other side of the environment. Chapter 8 concludes the thesis with the goals achieved and also a discussion about the avenues through which the current work could be extended.

CHAPTER 2

Although pedestrian movement is extremely common, modelling of pedestrian movement is an extremely difficult topic. Simulation has become an indispensable tool to study pedestrian movement in detail. Several models have been established to model crowd movements under different scenarios, and each has their own advantages, as well as deficiencies. In this chapter a brief description of various pedestrian simulation and modelling approaches is provided. After that a brief account of agent based modelling (ABM) and how pedestrian simulation can be performed within an ABM is provided. The specific objective in this study here is to model a situation where two groups of pedestrians are placed at two opposite sides of an environment. Their goal is to reach the other side of the environment traversing through the crowd of oncoming pedestrians. Some significant simplifying assumptions have been applied. All agents are assumed to be of the same size and moving at the same speed. The environment is also divided into a 2D grid of cells of equal size. Two models were developed for this study. The first one is a simple LEM and the other is modified ACO model. The following sub-sections provide a detail description of the algorithms used.

2.1 Pedestrian Simulation

The aim of the pedestrian simulation is to create virtual human beings in a given environment where they follow certain behavioural rules and make decisions to achieve a particular goal. Pedestrian simulation can be broadly divided into two different modeling approaches. The first approach is known as the macroscopic modeling [26]-[28] and the second approach is known as the microscopic modeling [29]-[30]. Macroscopic models do not deal with individual pedestrian characteristics and their features. A macroscopic model treats the whole crowd as a single entity and tries to define the movement of the pedestrian flows with differential equations. The pedestrian movement is analogous to the flow of fluids and gases. Macroscopic approaches include models such as fluid dynamic models [27]-[28]. A microscopic model on the other hand increases the granularity of the pedestrian movement. In microscopic models, the pedestrians are considered as individuals and the behavior, actions and decisions of each pedestrian is taken into account. A change in the behavior of pedestrian is based on their interaction with the environment and other agents. This alteration in the behavior will bring changes in the pedestrian dynamics as a whole. The microscopic model is more realistic in its nature than the macroscopic model. The microscopic approaches include social force models [1]-[3], cellular automata models [4]-[6] and agent based models [10]-[13]. In this subsection a brief description of each of the modelling techniques is provided and compared with the LEM and ACO based simulation approach.

A third type of modeling approach known as the mesoscopic model [10]. It is the mixture of microscopic and the macroscopic model. In this model, groups of pedestrians are modelled, so that the behaviors of groups are taken into account rather than considering either

individual pedestrians or the whole crowd as an entity. Mesoscopic models lies between the microscopic and macroscopic models. In this section, an investigation between the different approaches taken in the microscopic and macroscopic modelling is overviewed.

2.1.1 Fluid Dynamic Models

One of the earliest pedestrian models using macroscopic notions is based on a fluid dynamics model [27]-[28] which tries to model the whole crowd. This approach does not take individual pedestrians into account and describes the whole crowd with fluid dynamics equations. In this form of simulation, the pedestrians may be divided into groups having different types of motion. The pedestrian with each type of motion is characterized by several quantities such as their position, velocity and their intended velocity. So in a particular area, a population of pedestrians having the same kind of motion characteristics is observed and the motion of that group is governed accordingly by differential equations. Through the differential equations, the mean velocity, the velocity variance and the spatial and temporal density is obtained. The resulting equations that are obtained show similarities with the fluid dynamics equations. There are also some additional details that attempt to capture pedestrian intentions and interactions. Overall, in this type of pedestrian modelling, the pedestrians are modelled as a group and not as individuals. It is not very realistic in nature compared with a microscopic model but it has analytical and computational advantages. Both the LEM and ACO models are different from this modelling technique in that in LEM and ACO, each pedestrian is modelled individually, each having their own characteristics. So both the LEM and ACO models are essentially a microscopic model with much finer granularity than a fluid dynamics macroscopic model.

2.1.2 Social Force Models

This is a very early microscopic model. Social force models treat pedestrians as individual entities. The social force models [1]-[3] are mostly used in panic situations. Panic situations have different characteristics than normal pedestrian movement. In panic situations, individuals get nervous and travel faster than their normal speed. Their interaction with the environment and other pedestrians also changes as they become physical in nature rather than keeping a distance between each other. Using the social force model, modelling of a crowd is performed in both normal and panic stricken conditions. The pedestrians move towards their target driven by a force that is both physical as well as socio-psychological. These forces are combined to define movement of pedestrians in the environment in different situations. One of the major disadvantages of a social force model is the difficulty associated with accurately modeling complex socio-psychological interactions and behaviours. The LEM and ACO models also have the same notion in that the model is also based on the microscopic view.

2.1.3 Agent Based Modelling

Agent based modelling (ABM) [10]-[14] has been a prevalent modeling methodology for many years. However, it is gaining additional importance due to advancements in computing in general, and parallel and distributed computing for specific applications. As the LEM and ACO models developed here are based on the agent based system, the following subsection provides a brief description of agent based modeling and simulation.

2.1.3.1 Definition

An agent is an entity that is a part of a larger community and typically has a goal in mind. The agents are capable of carrying out their goals autonomously influenced by other agents. The two most distinguishing properties of agent based modelling is that the agents are able to perform their activities autonomously and they are part of larger community. Decisions of each agent may be influenced by other agents and vice versa, but they act autonomously. Agents are also a part of bigger community and the agents share the environment with other agents. Sometime agents have specialized roles or they might have a simple goal which is the same as other agents sharing the environment. The system of agents could be *centralized* or *distributed*. In a centralized agent system, there is usually a leader issuing commands to the subordinate agents. This is basically a military style control system. A distributed agent system is essentially a decentralized system where each agent is autonomous and works together with other agents, as could be seen in an ant system. The simulations carried out are based on distributed agent simulation process.

Agent based pedestrian model

Human behavior is extremely complex and often displays emergent phenomena. All agent based crowd simulations have some common features associated with them. In agent based crowd or pedestrian simulations, agents are defined by having certain properties. The environment is also defined which varies from one simulation to another. The agents make decisions individually based on their surroundings and may create an overall emergent pattern [31]. Agents make movement decisions autonomously based on their goals which may be influenced by their neighborhood within the environment. The

agents' decisions then modify the environment, which in turn influences the decision of agents in the future steps. This process is somewhat iterative in nature.

2.1.4 Cellular Automata Models

The LEM and ACO simulations are very similar to the cellular automata based simulation methods [4]-[8]. In a cellular automata based model, the whole environment is divided into rigid cells on a grid. The cells on the grid change their states based on certain rules. The change of cells' states creates a new generation of cells which actually depends on the previous cell states. In cell based pedestrian simulation, each pedestrian occupies a single cell or multiple cells based on the simulation. In cellular automata based simulation, pedestrians change their position on the grid, based on their neighborhood. Pedestrians move to their adjacent cells based on the condition of neighboring agents and cells. This causes the change of states of cells in the environment. In this kind of simulation, each pedestrian moves in the environment towards their goal with certain properties. While moving in the environment, the agent tries to avoid collision with both dynamic and static bodies. The movement of the agents in the environment may initiate an emergent phenomenon [32]. Different cellular automata based models have this characteristics in common [4][5]. However, the internal mechanism and rules on which the state changes varies from one to another. The LEM and ACO models used in this thesis also have characteristics similar to the cellular automata based models as the environment is also divided into rigid cells. Each pedestrian occupies a certain cell at a point of time and changes their position based on the neighbourhood condition. However, the environmental condition, simulation problem, and the rules used to change the state are unique to pedestrian

modeling. Apart from modelling rules, the LEM and ACO models have also been implemented on a parallel system, making the computation faster. Making the simulation faster using parallelization is a primary and central focus point. In the next sections, the theory behind the LEM and ACO models used for simulation purposes is described.

2.2 Least Effort Model

The Least Effort Model (LEM) [15] is a simple algorithm inspired from real world scenarios. In LEM, the agents move towards their respective target with a minimum deviation from their intended path of movement towards their target. A change in their path occurs only because of obstacles that arise in their movement towards their goal.

2.2.1 LEM Definition

In the LEM model, the environment is divided into rigid regular square cells. Pedestrians or agents are placed in the environment occupying a single cell at a point of time. For simplicity, it is assumed that all the agents are of the same size. At the beginning of a simulation, the agents have an initial position and also a target. They move in the environment toward their target and try to reach their goal or destination using the least effort. A situation is simulated where the agents are on one side of the environment and they are trying to reach the other side of the environment. When a pedestrian is present in the environment, then there are 8 neighbour cells as shown in Figure 2-1.

The movement of the pedestrian is dependent on distance of the cell which is nearest to the target. As such, the cell which is nearest to the target has the highest probability of becoming occupied in the next time step, and the cell which is farthest from the target

will have the lowest probability of becoming occupied by the pedestrian in the next time step. This is the basic mechanism which makes the agent move along a path which would take the least effort, defined as the shortest distance to the target. In reality, pedestrians are free to take a path which is not necessarily nearest to the target and so this randomized process is incorporated in the model. In effect, a pedestrian chooses a cell that is closest to the target.

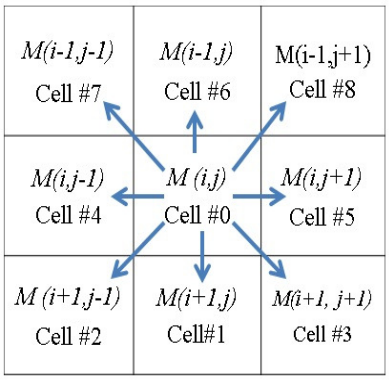


Figure 2-1: Neighborhood of pedestrian M placed in central Cell #0

Prior to making a decision, each pedestrian determines the distance of each unoccupied adjacent cell from the target and then ranks them according to their distance. The lowest rank denotes the adjacent cell which is nearest to the target (highest probability of being selected) and the highest rank is the adjacent cell which is farthest from the target (lowest probability of being selected). After that a biased random number is generated to decide which cell to select. The agent then moves to the cell chosen.

$$C_i = (1 - n_i)\{D_{min}/D_i\} \qquad (2\text{-}1)$$

$$D_{min} = Min(D_i),\ n \in \{0,1\},\ D_i \neq 0$$

In equation (2-1), D_i is the distance of all the neighbouring 8 cells from the target and D_{min} is the minimum D_i value. However, this distance is only calculated for empty cells as n_i becomes 1 occupied cells and C_i evaluates to zero. So, essentially C_i gets calculated for the cells which are empty but becomes 0 for the occupied cells. Then all the calculated C_i are ranked according to their distances in ascending order. A random number is generated for each of the agents to select a C_i. In order to generate a random number, a normal distribution is used with a mean value of 0 and a standard deviation of 3 was used although other distributions could also have been used as well. All the negative numbers generated from the normal distribution are suppressed. By using the above configuration, weighted random numbers from 0 to 7 gets generated.

Before making the calculation of the $C_{i,}$ a check is performed to determine whether the agent is already on the target column and if the cell in the next row is empty. If it is empty, then there is no further checking done and the agent makes a move forward towards its target.

As mentioned earlier, in this model the pedestrians are placed on one side of the environment and try to move to the other side of the environment. To achieve this goal the traditional LEM is modified and the target is made floating with only the opposite side being the target. As such, whenever they are in a certain column that is retained as their target column unless they change their column. This is the modification that has been considered to fit the LEM model for bi-directional pedestrian movement where reaching the other side has the highest priority.

2.3 Ant Colony Optimization

The second algorithm is based on an analog to ant colonies. Ant colony optimization (ACO) [16]-[18] was first introduced by Marco Dorigo in 1991. ACO is a metaheuristic algorithm which is inspired by the foraging behaviour of ant colonies. Metaheuristic algorithms have proved useful in solving several Combinatorial Optimization (CO) [33] problems. In CO problems, there is always a search space from where the solution is obtained. There is also an objective function(s) with a set of constraints. One needs to find the best solution satisfying all the constraints. In CO based problems, the problem space is discrete from where the best or optimal solution is obtained. Combinatorial optimization problems are typically extremely complex. Finding out an optimal solution may require an exponential computation time making them inefficient even for small problems. To reduce the computation time in solving many CO based problems, metaheuristics algorithms are often used. Examples of CO based problems are travelling salesman problem, knapsack problems, shortest path problems, etc., with the former being difficult to solve and the latter being simple. In recent years, metaheuristic algorithms have emerged as successful alternatives for solving difficult CO based problems. There are several metaheuristic algorithms that have been proposed, such as Ant Colony Optimization (ACO), Evolutionary Computations [34], Simulated Annealing [35], and Tabu Search [36]. With metaheuristic algorithms, often an acceptable or near global optimum is obtained in a short period of time by maintaining a balance between two processes known as *intensification* and *diversification* [37]. One of the most successful metaheuristic algorithms is ACO, which is inspired by biological ant behavior. In the implementation here, pedestrian movements are modeled after a modified ACO algorithm.

2.3.1 Ant Colony Optimization Discussion

ACO is based on the observed foraging behavior of ants. Ants are social insects living in colonies and they communicate with each other through a process called *stigmergy*. Pierre-Paul Grasse first coined this term after observing the effects of the stimuli given to some species of termites [38]. Termites live in colonies and after providing them with the stimuli, it further stimulates the termite operations. Essentially, the performance of the workers in the termite colonies is stimulated by the performance achieved so far. In *stigmergy,* communication occurs between the insects by modifying the environment. The same *stigmergy* process also takes place in the ant colonies. Ants, like termites, live in colonies. They travel to their food source and get back to their home by depositing a substance on the ground known as *pheromones*. While initially searching for food, ants start moving in arbitrary directions depositing pheromones. These pheromones act as positive feedback for the following ants and they take the route that has a higher pheromone concentration. The predecessor ants also in turn deposit pheromones, increasing the pheromone concentration. Pheromones evaporate over time if further deposition does not occur. As the ants start to forge arbitrarily in different directions, there are many paths of varying lengths. The ants taking the shorter trails will come back from the food source sooner, increasing the pheromones concentration along the trail. Gradually the following ants opt for the trails having higher pheromone deposits. So the pheromones on the shorter trails prevail, while for the longer trails the pheromones evaporate and disappear. This foraging behaviour of ants is the main source of inspiration behind ACO. An important application of ACO is to the Travelling Salesman Problem (TSP), where there are a set of cities and distance between them is provided, with the goal being to find the shortest

route to travel to all of the cities, visiting each city exactly once. This can be solved in a near optimal fashion and very efficiently using ACO [39]. The cities and the distances between them are typically represented by graph vertices and edges respectively. In ACO for TSP, ants are placed on the vertices of the graph and their movement to the next vertex depends whether it has already been traversed and also on the pheromone deposition on the edges of that vertex. The artificial ants build their individual solution but their decision is influenced by the pheromone deposition of other ants. After all the ants move to their next location the pheromone values are modified so that in the next iteration ants could choose shorter trails. After a point in time shorter trails with higher pheromone counts persist and the longer trails disappear. There are various forms of ACO but the basic concept remains the same. In the next section, the ant colony system for the TSP is presented in detail, as it is the basis for solving the pedestrian movement problem at hand.

2.3.2 The Ant System for the Travelling Salesman Problem

The Ant System (AS) used for solving the (TSP) can be divided into two distinct stages. The first stage is known as the *tour construction* phase where the ants build toward a solution. In this phase, there are n distinct cities which can be represented as nodes in a graph. In the problem considered, all the cities are connected and so the graph is fully connected. Initially m ants are placed on each city randomly. Ants build the solution by moving to the other cities. Suppose an ant k is placed in city i and the transition probability of the ant moving to the city j is given by

$$P_{ij}^{k}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{il}(t)]^{\alpha}[\eta_{il}]^{\beta}}, j \in N_i^k \\ 0, j \notin N_i^k \end{cases} \quad (2\text{-}3)$$

P_{ij}^{k}: The probability of the k^{th} ant to go to city j while currently at city i on the t^{th} step.

η_{ij} : Heuristic value from city i to city j. Generally $\eta_{ij}=1/d_{ij}$, where d_{ij} is the distance between city i and j.

τ_{ij} : Is the amount of pheromone deposited in the path of city i and j.

α,β : These are the two parameters that control the relative weight between the pheromone trail and the heuristic value of the distances between current and next probable city respectively.

N_i^k : Represents the feasible neighborhood of cities that ant k could visit at that point of time. This set of cities should not be those visited by the ant in any previous steps. The probability of visiting any cities other than these cities in the set is 0.

After the ants build a partial solution and construct their path, the *pheromone update* stage is performed. In the pheromone update stage the pheromones are lowered on all the nodes or edges by a constant factor. The lowering of pheromone values helps to avoid getting stuck in local minima. This is known as pheromone evaporation given by

$$\tau_{ij} \longleftarrow (1-\rho)\tau_{ij} \quad (2\text{-}4)$$

Where $0 < \rho \leq 1$ is the pheromone evaporation rate. After the evaporation, the pheromones are deposited on the visited edges as given in (2-5)

$$\tau_{ij} \longleftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \quad (2\text{-}5)$$

Where $\Delta\tau_{ij}^{k}$ is the amount of pheromone deposited by ant k on the edge between city i and city j, defined as in (2-6)

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{L_k}, & \text{if the edge (i, j) belongs to the k}^{\text{th}}\text{ ant} \\ 0, & \text{otherwise} \end{cases} \quad (2\text{-}6)$$

L_k is the length of the tour T^k built by the k-th ant and is obtained by the summing up the lengths of the edges in the tour.

The two most successful variants of AS are Max-Min Ant System (MMAS) and Ant Colony System (ACS). In MMAS, only the best ant updates the pheromone count on the trail. However, the count of the pheromone is bounded. After the pheromone count on a particular edge reaches a certain value, further updates stop. The pheromone update is as follows:

$$\tau_{ij} = \left[(1-\rho).\tau_{ij} + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{max}} \quad (2\text{-}7)$$

τ_{max} and τ_{min} marks the upper and lower bound of the pheromone count.

$$\Delta\tau_{ij}^{best} = \begin{cases} \frac{1}{L_{best}}, & \text{if (i, j) belongs to the best tour} \\ 0, & \text{otherwise} \end{cases} \quad (2\text{-}8)$$

L_{best} denotes the length of the tour of the best ant.

In ACS, the main contribution is the introduction of the local pheromone update along with the final update at the end of the process. The local pheromone change is as follows:

$$\tau_{ij} = [(1-\varphi).\tau_{ij} + \varphi.\tau_0] \quad (2\text{-}9)$$

Where $\varphi \in (0,1]$ is pheromone decay factor and τ_0 is the initial pheromone value.

2.3.3 Ant Colony Optimization Application

Ant colony optimization has been used in several applications. One of the most common applications is its use in solving *NP*-hard problems [40]. In *NP*-hard problems, getting an optimal solution takes an exponential time. *NP*-hard problems include routing based prob-

lems [41], assignment problems [42] and scheduling problems [43]. In some of the *NP*-hard problems, ACO proved to be a viable method [39].

In simulations performed, the AS used for the TSP is used to model the movement of the pedestrian in the environment. The primary motivation behind using ACO algorithm for the simulation is to replicate some sort of visual indication by the predecessor for a following pedestrian. The implementation and the results obtained by using ACO based model is discussed in detail in the later sections.

In this chapter, a literature review on pedestrian modelling and the methods used to simulate them was provided. A discussion on the algorithms used for simulation purposes is also discussed. Later chapters present the model description and the methods used for implementation on a GPU. A detailed discussion concerning the GPU architecture and the methods adopted for GPU optimization is also presented. Finally the result obtained by using ACO and LEM based models is also provided.

CHAPTER 3

This chapter provides a comprehensive description of the simulation models. While describing the algorithms, the modifications for the specific problem addressed here are discussed.

3.1 Model Description

The CPU simulation is written in C on Windows and Microsoft Visual Studio 2010 as the Integrated Development Environment. An NVIDIA GPU was used for the GPU implementation; the CUDA programming language is used. The details of the NVIDIA GPU architecture and CUDA programming interface are provided in Chapter 4.

3.1.1 Agent and Environment Characteristics

In this subsection, the characteristics of the agent and environment parameters are discussed. The environment is divided into 2D grid cells of equal size and the environment is chosen to be square in size. To perform the simulation, the size of the environment is chosen to be of size 480x480, to accommodate a large number of agents. The size of the environment should be a multiple of 16. The reason behind such a selection is provided in the implementation Chapter, but is essentially a consequence of occupancy of the GPU. All agents are of the same size and move at the same velocity in all the simulations performed. At a point of time, each cell of the environment is occupied by a single agent. All

the agents have the same characteristics but they have different goals. The agents placed along the top of the environment have the end row of the bottom as their goal, opposite to case of the agents placed along the bottom, whose goal is the top row. In a single time step, they are free to move to one of the neighborhood cells based on its availability. For a particular simulation the number of agents remains constant. However, the number of agents increases in from one simulation to another. The number of agents present in the environment in each of the simulations is mentioned in the performance analysis section 6.1 and result analysis section 7.1. A sample environment is shown in Figure 3-1 which is a scaled down version of the target environment. The pedestrians placed along the top are labelled with a 1 and the pedestrians along the bottom are marked with a 2. The empty cells are marked as 0.

The number of pedestrians both in the top and bottom of the environment is user defined and declared at the beginning of the simulation. The initial placement of the pedestrians is random. They are however limited up to a certain number of rows upon initialization. In Figure 3-1, a sample environment of size 16x16 is illustrated. The agents placed along the top are labeled *Top Pedestrians* and agents placed along the bottom are labeled *Bottom Pedestrians*. In Figure 3-1, there are total of 58 agents in the environment, 29 agents along the top and 29 agents along the bottom.

An approximately equal number of pedestrians is maintained along the top and bottom portion, which is not mandatory, but is done to maintain balance in the simulation. In effect, this makes it easier to confirm that the simulation is valid, as this constraint should on average provide symmetry between the agents traversing up as well as down. The initial placement of both top agents (marked by 1) and bottom pedestrians (marked by 2) are

random following a uniform distribution. Their initial placement is limited up to a certain number of rows which is 3 (Rows == 3) in Figure 3-1. The number of rows for the initial placement of the agents is also user defined. Though it is not mandatory, the rows up to which the agents would be placed initially is also kept the same for both the top and bottom placed agents to maintain uniformity. As mentioned previously, the target of an agent is to reach the other side of the environment. So the target for a particular type of agent is to reach the last row of the opposite side. For an agent placed along the top, the target is to reach 16th row and for an agent placed along the bottom the target is to reach the 1st row. In practice, the objective for the agents is to get as far as possible to the end row of the opposite side unless no further movement is possible.

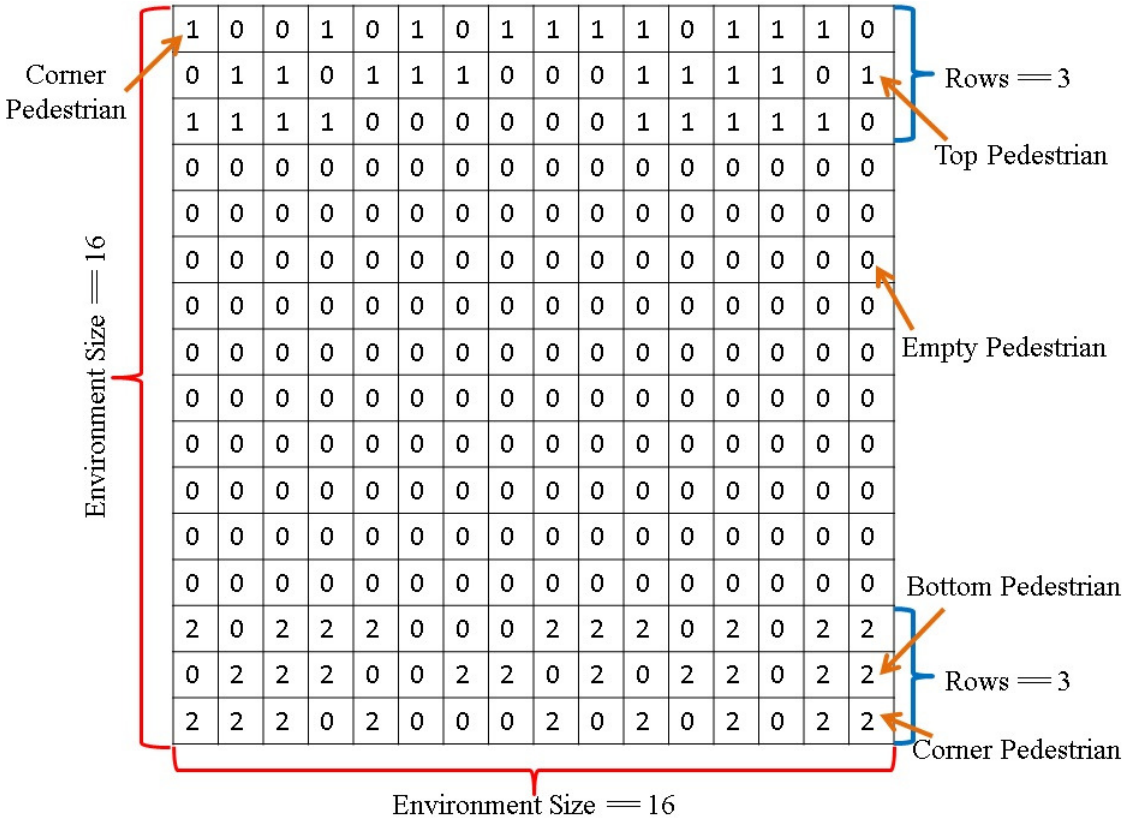


Figure 3-1: Sample original environment of size 16x16

Usually a pedestrian placed in an environment is surrounded by 8 neighboring cells. As shown in Figure 3-2, a pedestrian *M* is placed in the central *CELL #0* with a coordinate of *(i,j)*. The agent surrounded by 8 neighborhood cells starting from *CELL #1* up to *CELL#8*.

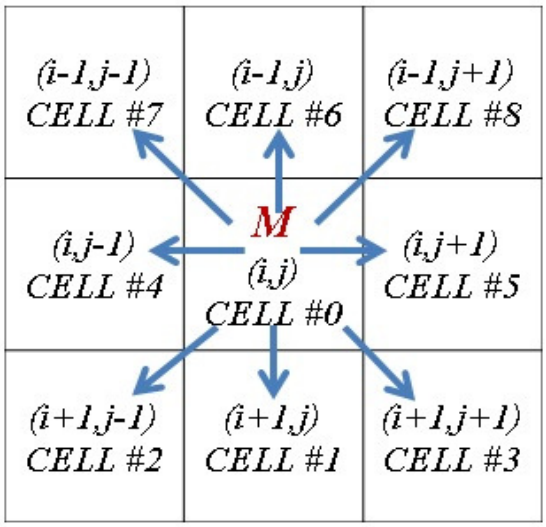


Figure 3-2: Pedestrian *M* placed in a central cell surrounded by 8 neighborhood cells

However, the agents placed on the border or corner positions are not always surrounded by 8 cells. In Figure 3-1, there are agents in both the top and bottom positions which are labeled *Corner Pedestrians*. For those agents the total number of available neighborhood cells gets reduced due to their corner positions. The number of missing cells and their co-ordinates depend on the position of the corner agent. Other than the corner and edge agents, they are surrounded by 8 neighborhood cells. A pedestrian is free to move to any of the neighborhood cells based on the cell's availability. The movement of agents in the environment depends on the algorithms used for their simulation. As mentioned two algorithms, LEM and ACO, are utilized for the simulation. There is one difference between the implementation of LEM and ACO based model. The following subsections describes in detail both the models used.

3.1.2 LEM Based Simulation

In Figure 3-2, a typical scenario is depicted where a pedestrian *M* is placed in the central cell (*CELL #0*) with a coordinate *(i,j)* and it is surrounded by 8 cells. Pedestrian *M* is free to move to any of the surrounding cells at the next move. Their movement in the next step is guided by LEM algorithm. There are certain modifications that are incorporated into the LEM. As mentioned, the simulation concerns two groups of pedestrians placed at opposite sides of an environment, and their goal is to reach the opposite side of the environment. In this type of simulation, the forward cell of a pedestrian has the highest priority. In Figure 3-2, pedestrian *M* placed in *CELL #0* is moving from top to the bottom, the highest priority is given to the forward *CELL #1*. It would have been *CELL #6* if the pedestrian is moving from the bottom to the top of the environment. While making a decision, the pedestrian first checks the availability of the forward cell. If the forward cell is empty, then the pedestrian decides to move to that cell without any further calculation. Through this decision, an agent is able to move forward unimpeded towards the opposite end of the environment. If the forward cell is occupied, then the agent checks the availability of the surrounding cells. If none of the surrounding cells are found empty, then the agent stays in the current position. Moving to a surrounding cell depends on the cell's availability. The movement of an agent to a surrounding cell in the LEM based model is guided by the rule of equation 2-3. Among the remaining seven cells, an agent first checks the availability, identifying the cells that are empty, carries out the calculations of equation 2-3 for the surrounding empty cells and based on their distance from the target, and ranks them in the ascending order. As mentioned in the LEM model description, a pedestrian does not always select the cells having a least distance from the target. In a re-

al world scenario, pedestrians also elect paths that are not optimal. The surrounding empty cells are first determined by scanning the neighborhood. A biased random number process indicates the next cell to be chosen. In figure 3-3, pedestrian *M* is placed in central CELL #0 with two empty neighborhood cells (CELL #2 and CELL #8). The distance d_1 of CELL #2 from the target is less than distance d_2 CELL#8. After scanning the neighborhood, the distances of the empty cells from the target are kept in an array. The distances are then normalized. A random number is generated from a normal distribution with mean value of 0 and standard deviation of 1. The random number signifies a particular cell to be selected. The negative portion of the distribution is converted to a very small number which helps to select the cell having the least distance from the target. A particular side has given more bias than other side to the agents placed on a particular portion of the environment. In our case we have given right hand side better preference than the left as the agents most of the time are more inclined towards their right. That is in the case of a tie, the agent will tend to the right side.

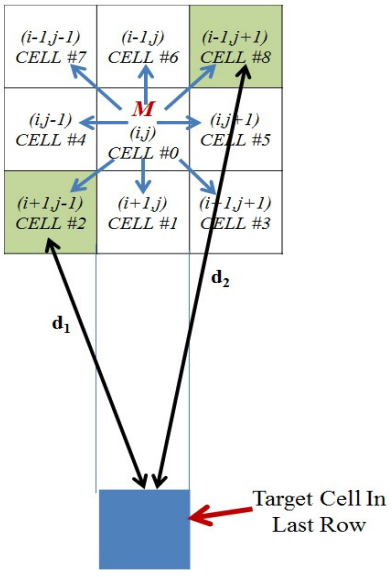


Figure 3-3: Illustration of LEM calculation for pedestrian M

3.1.3 ACO Based Simulation

To emulate the bidirectional pedestrian movement, a variation of ACO which is used to solve the TSP was also used. The ACO algorithm was modified for this specific bidirectional pedestrian movement scenario. In this simulation, the agents are also placed in two groups along the top and bottom of the environment. The initial position of the agents are uniformly and randomly distributed but kept limited to certain number of rows, as was done in the LEM based simulation. The number of agents that would be placed in the environment and the rows up to which they are initially placed is user defined. The decision making process in general is similar to that of the LEM. As shown in Figure 3-2, an agent M is placed in *CELL #0* and is free to move to any of the surrounding cells based on their availability. However, an agent's target is to reach the other side of the environment which gives the forward cell the highest priority. So if M is considered to be an agent placed along the top and moving to the bottom of the environment then *CELL #1* has the highest priority. For an agent placed along the bottom moving to the top, the highest priority cell would be *CELL #6*. If the forward cell is available, the agent moves to the forward cell without any further calculation. This process allows an agent to move towards its target without any further path deviation in the shortest time possible. However, if the forward cell is found occupied, then the agent looks for the other available neighbourhood cells. The availability of the cells depends on the position of the agent and also on the agent density in the immediate neighborhood. As mentioned earlier, if an agent is placed in the corner or on the edges of the environment, then the number of available cells is modified. After checking the availability of the neighborhood cells, the agent has knowledge about the number of cells available. To choose one from among the available

cells, the ACO algorithm is used. In this kind of simulation for a particular agent, it is desired to reach the goal traversing as few cells as possible and also to follow a predecessor agent who is able to move forward through the crowd. This would provide a real world scenario where pedestrians try to reach their destination with as little effort as possible and making as little deviation from the current path as possible, yet taking cues from the pedestrians ahead of them. In a practical sense, this model may allow for the emergence of streams, which is a widely recognized form or self-organization observed in real crowds [44].

The TSP is basically a routing based problem where the salesperson is to visit all the cities once. The problem is to find the shortest route possible to perform this job. So in solving the TSP problem using the ant system algorithm, the heuristic value is the distance between the cities. However, in the ACO based simulation, the heuristic value (η_{ij} in 2-3) is replaced by the distance (d_{ij} in 2-3) of the neighboring cells of the agents to the target, which is the end row of the opposite side. After getting the neighborhood distance, it is inverted ($\eta_{ij} = 1/d_{ij}$ in 2-3) and then it is raised to a power (β in 2-3). The neighborhood cell farthest from the target has the lowest value and the nearest ones have the highest value. So the agents are able to move in the environment making effort saving decisions. In the original ant system model for TSP, ants are not able to move to the cities that are previously traversed. The present simulation does not have any cities or cells where the agents can be present only once. After gathering the knowledge of the neighboring cells availability, the inversed distance or heuristic value and the pheromone value of the particular empty neighboring cell are multiplied. This is essentially the computation mentioned in Equation (2-3). In the above computation, both the distance of the neighborhood

cell and the pheromone value contributes in the agent's decision making process. This accomplishes the tour construction phase of the agent movement.

After all the pedestrians finish with the tour construction phase, the pheromone update stage is performed. Pheromones get deposited on a particular cell whose value depends on the tour of that particular pedestrian. Pheromone evaporation is also carried out to avoid any local minima situation. Both the pheromone deposition and evaporation take place following Equations (2-4) and (2-5). This whole simulation is carried out a fixed number of time steps for each experiment.

In this Chapter the environment and the agent's parameters used during the simulation purposes were briefly presented, including how the LEM and ACO based algorithms were modified for the simulation purposes, in order to assist in the efficient implementation. This chapter explained how the ACO and LEM based algorithms are modified to fit into the specific simulation purposes addressed here.

CHAPTER 4

In the applications discussed here, a GPU is used to simulate the pedestrian movement to expedite the otherwise compute intensive simulation. There are several factors that come into consideration while programming a GPU. These factors are very much related to the GPU architecture and design. For the simulation purposes, an NVIDIA GPU is used and CUDA is used as the programming tool. In this chapter, a detailed description of the GPU architecture is provided to better understand the rationale behind the implementation of the algorithms.

4.1 Introduction

The name of the parallel computing platform and programming model for NVIDIA is known as Compute Unified Device Architecture or CUDA [25][46]. The CUDA programming platform is based on Graphics Processing Unit (GPU) [23]-[24] technology. Before the advent of CUDA technology, for general purpose computing GPUs were programmed using graphics APIs. The disadvantage of this process was that the programmer needed to have a very good knowledge of graphics processing. NVIDIA expanded this limitation by extending GPU programming for general purpose computing through CUDA C. The first CUDA capable device was based on the G80 architecture [45] re-

leased in 2006. This section introduces the basic concepts related to CUDA programming.

The code to be executed by the CPU is referred to as the *host code* and the code executed on the GPU is referred to as the *device code*. The code on the GPU device is performed by thousands of lightweight CUDA threads. The CUDA threads are lightweight because their execution is controlled by hardware. The threads are managed and scheduled at the hardware level. The programmer has little control over the threads and their order of execution, as this is handled in hardware. The threads in a GPU are executed in parallel by the multiprocessor units each containing a number of processor cores. The CUDA programs are portable between the compatible versions of GPU irrespective of the number of processor cores. Threads are grouped into blocks, each containing same number of threads. The blocks are arranged into grids. The threads and the blocks can be arranged from 1D to 3D depending on the application. The blocks are assigned to the multiprocessor which is hardware scheduled, which makes CUDA portable across hardware versions and able to execute regardless of the processor count or resources of the GPU.

CUDA threads operate in a group of 32 threads as a single execution unit known as *warp* [23]. The same instructions are executed for the 32 threads in the warp. If any thread within a warp goes through a different execution path, then a phenomenon called thread divergence occurs. This thread divergence is one of the most important factors behind making the program serial instead of parallel. A conceptual view of a CUDA block and threads is shown in Figure 4-1.

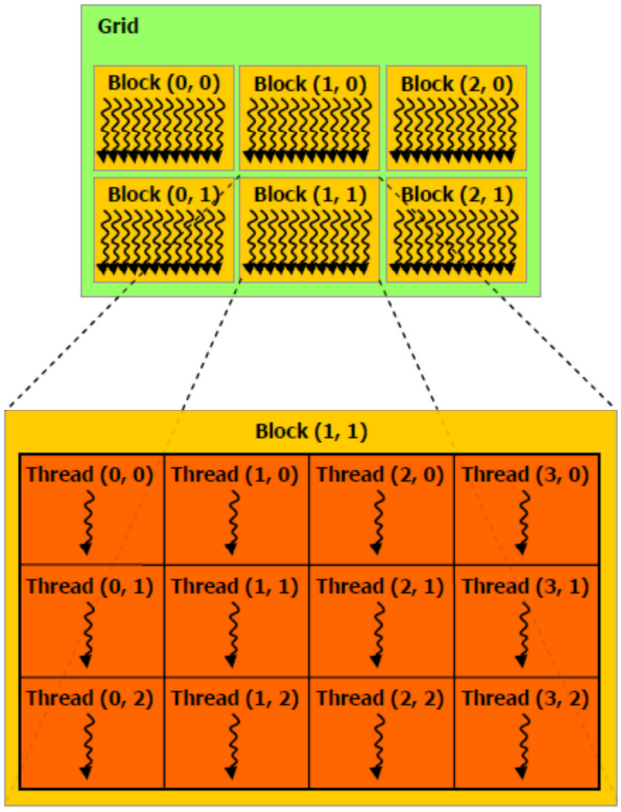


Figure 4-1: CUDA Block and thread architecture [25]

4.2 CUDA Architecture

For the simulation undertaken here, a NVIDIA GTX560ti model is used which is based on FERMI architecture [45]. The GPU card acts as a coprocessor to the main processor in the system. The GPU card is connected with the system through PCI-Express bus. The latency and width of the bus determines the rate of transfers from the host main memory to the device memory. In the next subsection, the GPU architecture is discussed.

4.2.1 GPU

The GPU contains a hierarchy of processors which is the main driving mechanism behind parallel implementation of the threads. There are a series of processors which sit in the

top of the hierarchy known as the Streaming Multiprocessor (SM). Each of these Streaming Multiprocessors consists of a number of Streaming Processor (SP) which are also known as CUDA cores. The GPU used for the simulation purposes has a total of 448 CUDA cores. The particular GPU version used contains 32 SPs in each SM, for a total of 14 SMs. The kernel function launched on the GPU is same for all the threads with a varying thread index number. There is a GigaThread scheduler which distributes the blocks to the SMs thread schedulers. The 16 load/store units loads the source/destination addresses for a half warp [45].

Each SP has its own Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU). The FPU supports both the single and double precision operation. FERMI supports fused multiply-add (FMA) [45] instructions for both the single and double precision floating point operations. FMA instructions reduce precision loss due the rounding or truncation. Each of the SMs contains Special Function Units (SFU) which contains the hardware support to perform the sine, cosine, and square root functions. The SFU functions independently and so the SPs can work on other instructions in the meantime. The FERMI has two warp and two dispatch units which enables each SM to work on instructions from two warps simultaneously. The CUDA architecture is described as single instruction, multiple threads (SIMT) where a warp of 32 threads execute simultaneously when the conditional instructions do not cause the threads to diverge, serializing the application. However, the threads can also work independently if they do diverge. In Figure 4-2 the FERMI architecture is shown. In the next subsection, a brief account on the memory options available on GPU is discussed.

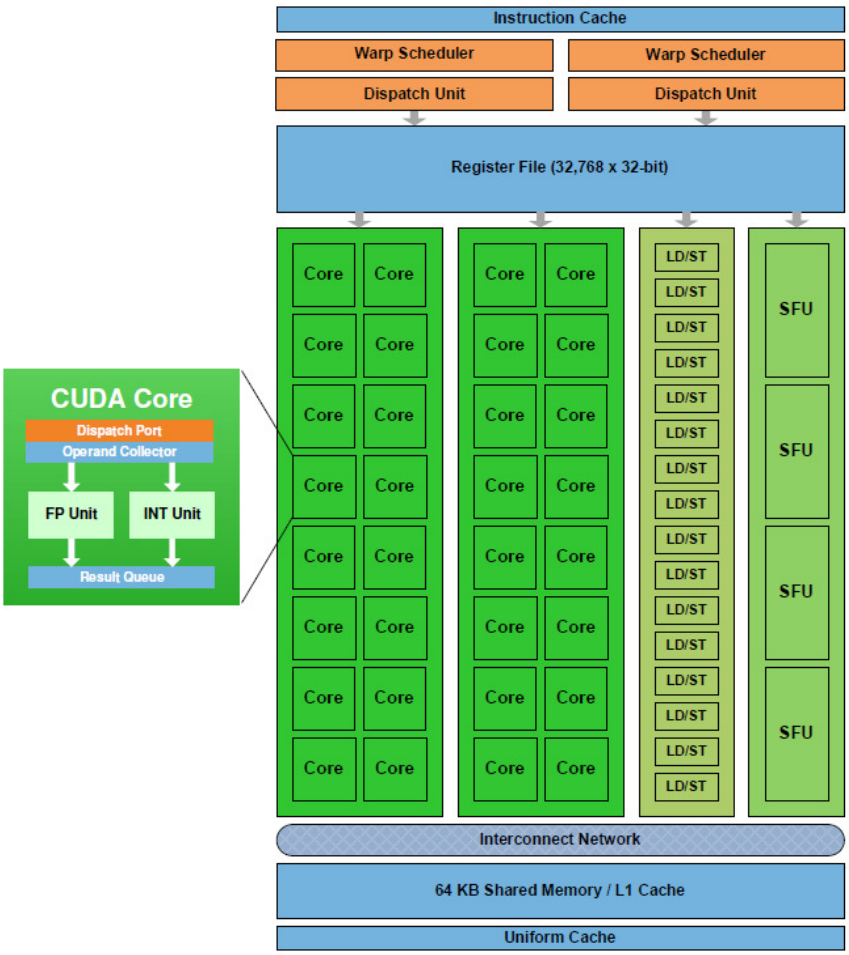


Figure 4-2 FERMI Architecture [45]

4.2.2 GPU Memories

The GPU is connected with the main system through the PCI express slot. Data first gets transferred from the CPU main memory, also known as the host memory, to the GPU memory. One of the primary goals for increasing the GPU operation is to reduce the data traffic between CPU main memory and GPU memory. So before any operation, all data is transferred and then the GPU starts working. The FERMI GPU has a hierarchy of memories each with a different purpose. A conceptual view of the memory architecture is provided in Figure 4-3

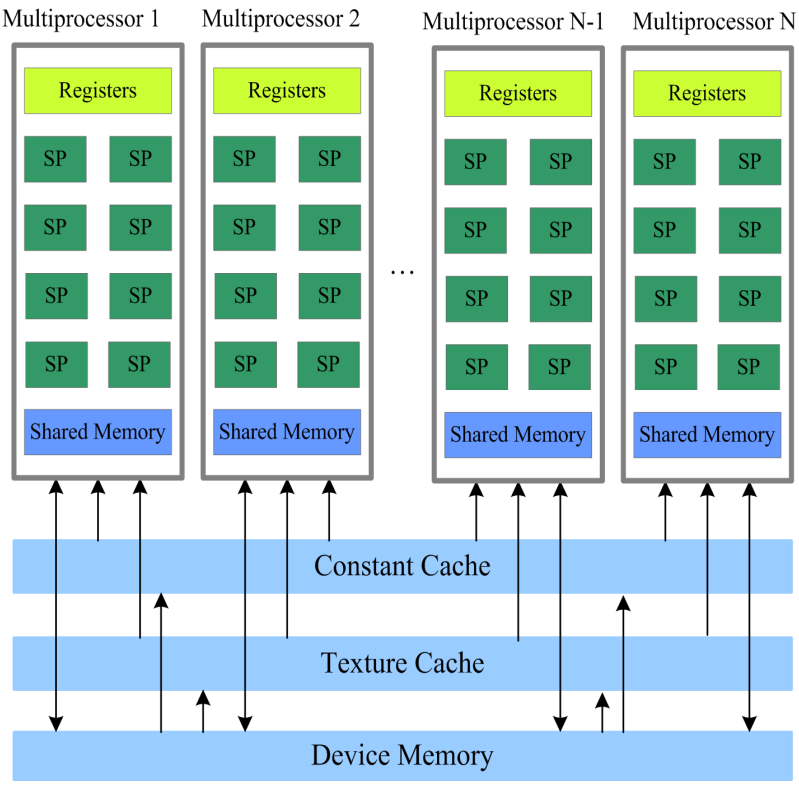


Figure 4-3 Memory Architecture Overview [47]

The main memory present in the graphics card is also known as the global memory. It is located off-chip and the slowest among the other kind of memories. The GPU that has been used for the simulation purposes has an off-chip global memory of size 1280 MB GDDR5 RAM. The reads and writes from the global memory by the multiprocessors is incoherent as the multiprocessors works independently. Yet, the read and write from the GPU main memory can be made coherent through programming using a technique known as memory coalescing [23]. In this process, the threads fetch the data from the co-alesced memory addresses reducing the number of accesses to the main memory. There is a level 2 cache of size 768 KB that performs load, store and texture operations. GPUs have another memory used for texture in graphics processing. This memory is known as the texture cache. The data in the texture cache is read only and as the data is cached the threads are able to fetch the data in much fewer clock cycles. Similar to the texture cache

another memory is also present in the GPU which is known as the constant cache. Like the texture cache, data in constant cache is also read. The data in the constant cache can be accessed faster as the data doesn't change during the program runtime. The GPU used for the simulation has a constant memory of size 64 KB. There is faster form of on-chip configurable memory type known as the L1 cache. As shown in Figure 4-2 each of the multiprocessors or SMs have their own shared memory. The GPU version that is used has 64 KB of shared L1 cache memory which is configurable to 48 KB and 16 KB. Depending on the program, either shared or constant memory could be configured as 48 KB or 16KB and the other one would get the rest of the memory portion. Data in this memory is shared between all the threads in a single block. The data is first brought in from the global to the shared memory and then the threads in the block work on this data portion. This reduces the global memory traffic from the off-chip global memory to the on-chip shared memory and increases the GPU speedup significantly. The data in the shared memory is available while the block is functional. The fastest form of memory is the register which is private to each thread. Each of the threads has a numbers of registers. The automatic variables within the CUDA program are kept in the registers. Although the total number of registers available is large, their number is limited in the case of each thread to maintain 100% occupancy. The maximum number of registers that can be used by each thread can be obtained from the CUDA occupancy calculator [48]. The registers are active as long as the thread is functional. The maximum numbers of registers that can be used by a thread is 20. If the arrays declared within the CUDA program or the data to be stored in the registers is more than the permissible amount, then the additional data are stored in a portion of global memory which is known as local memory. The local memory

suffers from the same latency as the global in the case of accessing the data. Data requirements being more than the permissible amount is known as register spilling.

4.2.3 CUDA Capability

Each CUDA capable GPU comes with a CUDA compute capability [23] version. The one used here has a CUDA compute capability of 2.0. This CUDA compute capability versions signifies a certain set of capabilities. As the compute capability version increases, the features available also increase. The CUDA program compiled with a lower compute capability will not work for a higher compute capability version. However, the compute capability version is backward compatible. As the compute capability version increases the resources available also increases. A GPU with a higher compute capability has greater number of registers available for each thread, a greater number of threads can be launched in each block, more blocks can be present in the grid, a greater amount of shared memory is available for each block, and an increase in speed and accuracy of double precision floating point operations as well as faster atomic operations.

4.3 CUDA Parallel Programming

CUDA programming language is an extended version of prevailing computer languages like C, C++ and FORTRAN. However for the entire simulation purposes CUDA C has been used. In the case of C, CUDA provides some additional features and application programming interfaces (APIs). CUDA programs are compiled using CUDA NVCC compiler [49]. CUDA 5.0 was used for the programming purposes here. In this section the CUDA programming features that are used in pedestrian movement application are

discussed. The next subsection provides a brief account on the qualifiers used in CUDA programming language.

4.3.1 Function Qualifiers

The most primary is the __global__ function qualifier which is called from the host portion of the program. The function defined under this qualifier is then executed on the GPU device. The return type of this function is void. The function declared under __global__ qualifiers are known as kernel functions on the CPU side. When these kernel functions are launched from the CPU side, a definite syntax is maintained. Inside the host function, the required numbers of threads are arranged into blocks and blocks are arranged to form grids. Then, the kernel function is launched with the thread and block configuration. Between the function name and the parameters of the kernel function on the CPU side, the thread and block configuration is specified. The format is as follows:

Function_name<<<blocks,threads>>>(Function_parameters)

After this is declared inside the host function, the kernel function is defined outside the main function under the __global__ qualifier. The format is as follows:

```
 __global__ void Function_name(Function_parameters)
{ ……..... ;
  …… ;
 }
```

The function pointers are mainly present in the function parameters which contain the address of the data placed in the global memory. Other than the using data stored in global memory, data placed in other memories can also be used.

Another function qualifier is the __device__ qualifier. The function defined under this qualifier is only executable through other functions defined on the device. This means

that this function is callable from other __device__ functions or from a __global__ function.

There is another function qualifier known as the __host__ function qualifier which indicates the function defined under this qualifier is executable only from the host machine.

4.3.2 Variable Qualifier

In this section the variable qualifiers available are discussed. The __device__ variable qualifier is used to define the variables declared in the device global memory. The variable declared under this qualifier is accessible by the device and host functions. The variables are actually valid until the program exists.

Another kind of variable qualifier is the constant variable qualifier. It is declared under __constant__ syntax and the variables declared under this qualifier stored in the constant memory. The data in the constant memory gets cached during the runtime and the program can access it at a much faster rate.

The variables declared under __shared__ variable qualifier stores the data in the shared memory. However, this form of variable can only be declared inside device functions. The variables have the lifetime of the block and are shared by all the threads in a block. As the variables in the shared memory are shared by all the threads then some kind of barrier synchronization is required to obtain the correct result. If barrier synchronization is absent, then the final result may be erroneous if one thread ends earlier than another. Barrier synchronization is performed by using __syncthreads().

There are automatic variables also that can be declared inside a thread and are private to each thread. The values stored inside the automatic variables can be different for all the threads. Automatic variables are stored in registers. However the number of registers

available to each thread is limited and if the numbers of variables are more than permissible amount, then the additional variables gets stored in the local memory. The automatic variables have the lifetime of the execution of the threads. The arrays private to each thread gets stored in the local memory by default.

4.3.3 Built-in Variables

There are built-in variables defined in the CUDA programming language that are discussed in this subsection. One of the variables which is used in all the CUDA programs is the gridDim through which the programmer could define the block size and the dimension. The variable is of type dim3. Each of the blocks in the grid has a unique identification number. The blockIdx built-in variable is used to identify the blocks in the grid which is of type uint3. This is actually the index position of the block in the grid. The second important built-in variable is the blockDim which is again of type dim3. This actually indicates the block dimension defined by threads in the blocks. The index number of the threads in the block is given by threadIdx. The threadIdx is also of uint3.The warpSize built-in variable is of type int and contains the number of threads in a warp. It is 32 in the GPU used for the simulations.

Information regarding the hardware resources can be obtained through device query functions. The variable defined as cudaDeviceProp struct stores the information regarding the hardware resources. This struct variable address is sent as a parameter inside the cudaGetDeviceProperties() which returns the device properties. Through this, all the device information is obtained. This variable provides the number of multiprocessors available, global memory size, constant memory size, shared memory size, number of registers, maximum number of blocks in a grid and threads in a block and several other pieces

of information. The information that could be obtained from the device query is provided in [24].

4.3.4 Memory Related API

Through the CUDA memory operations, the global memory of the GPU is accessed. The host memory is dynamically allocated using the malloc operation. The global memory of the GPU can be allocated in the same way using cudaMalloc(). The address pointer of the variable and the amount of memory needed is passed as parameters through the cudaMalloc() function. The API returns an error code indicating whether the allocation is successful or not. The return code is cudaError_t. The cudaMalloc() API format is provided below:

```
cudaError_t cudaMalloc(void** variable, size_t variable_size);
```

The memory allocated using the cudaMalloc() function needs to be de-allocated after its use is over. The memory allocated using the cudaMalloc can be de-allocated using the cudaFree() function. The name of the device variable is passed as the parameter of the function. It is very important to release the memory to avoid any sort of memory leakage. As mentioned previously, the data is first copied from the host main memory to the device global memory and then the threads work on the global memory data. The copying of the data from the host to the global memory is performed using cudaMemcpy() function. In the cudaMemcpy() API, as the first argument, the pointer to the destination address is sent which could be in device global memory or in the host memory. Next the pointer to the source address is sent which could again be global memory of the device or host main memory. As the third argument, the size of the memory needing to be copied is

mentioned. The fourth argument actually indicates the type of the memory transfer. Five kinds of transfers could take place given below:

cudaMemcpyHostToHost: Transferring data from host memory to host memory.

cudaMemcpyHostToDevice: Transferring data host memory to device global memory.

cudaMemcpyHostToHost: Transferring data from device global memory to host memory.

cudaMemcpyDeviceToDevice: Transferring data from device memory to device memory.

cudaMemcpyDefault: Default memory copy based on virtual address space.

This API has a return value of cudaError_t. The cudaMemcpy() API format is provided below:

```
cudaError_t cudaMemcpy(void* dst, void* src, size_t count, enum cudaMemcpyKind)
```

dst: pointer to the destination memory address

src: pointer to the source memory address

count: number of bytes needed to be copied

cudaMemcpyKind: Type of transfer.

To transfer the data from the host main memory to the device global memory, cudaMemcpyToSymbol is used. The data can also be copied from the host memory to the constant memory of the device. The pointer to the device variable in the constant memory, the pointer to the host memory and the size of the memory that need to be copied is sent through the argument. As the return value, cudaError_t is sent.

Similar to memset() in the C, CUDA also has cudaMemset() API. Through this API the value in the device global memory already declared using cudaMalloc() operation can be

initialized. In the argument, the pointer to the device memory, the initialization value, and the number of bytes that need to be initialized is sent.

4.3.5 Atomic Operations

The threads in the GPU operate in parallel. When the threads operate simultaneously on the same memory location modifying the value stored in it, there would be a race condition and an erroneous value may be generated. To avoid this kind of condition, atomic operations can be used. If the CUDA threads perform the operation using atomic operations, then the memory position would be locked for the particular thread and no other thread would be able to access the same memory position. In the task based parallelism, atomic API atomicExch() have been used. The API reads a 32-bit or 64-bit word located in address whose pointer sent through the argument and then exchanges it with a value. The value to be exchanged is also sent through the argument. The function returns the old value that was replaced with the new value. The API format is provided below:

Return_type atomicExch(address_pointer, value)

There are other atomic operations used for different kind of operations like atomic addition or subtraction, finding the minimum or maximum in a group of values. There are also bitwise operations like atomic and, or and xor.

4.3.6 Random Functions

In the simulation, random numbers are generated by using APIs. The CURAND library [54] is used to obtain the random numbers. For the simulation purposes the APIs from the library is used to generate the random numbers from the normal and uniform distribution. To generate a random number, the random number generator state is initialized using the curand_init() API. A seed is also passed to the initializer API which also participates in

the initialization. The CPU real time clock is used as the seed to the random number generator. Each of the threads generates their own unique random numbers.

In this chapter, an overview of the NVIDIA FERMI GPU architecture was presented, followed by a detailed description of the CUDA programming language. Primarily, the APIs that were used in the programming purposes were discussed. There are also several other features of the CUDA programming language in use that are dependent on the application being implemented. The CUDA features discussed in this Chapter are helpful in understanding the implementation chapters described later.

CHAPTER 5

An NVIDIA GPU and CUDA C was used as the programming tool for the pedestrian model. This type of modelling can be performed using two forms of parallelism known as *task driven parallelism* and *data driven parallelism*. In a GPU, thousands of threads can be launched in parallel and the behaviour of the threads determines the kind of parallelism implemented. In the crowd model here, each pedestrian is independent of other pedestrians. All the agents move in the environment simultaneously and their decision depends on their surroundings. As such, each agent can be treated as a single CUDA thread. This form of parallelism is known as *task driven parallelism* where each thread has its own task.

A square 480 x 480 2D environment is used which is divided uniformly into cells. An agent occupies a single cell at a point of time and may move to one of the neighborhood cells based on their occupancy. In this case the CUDA threads can be launched on the environment cells. Each of the threads can operate on each environment cell which is essentially working on the data. This sort of parallelism is known as *data driven parallelism*. Each kind of parallelism used for the simulation has its own advantages and disadvantages. Task driven parallelism provides freedom in deciding the parameters like the environment size and total number of agents. However, task based parallelism is not suitable for GPU based simulation as numerous issues arises that impedes the parallel opera-

tion of GPU, including warp divergences, deploying atomic operations and underutilization of on-chip memory which are some of the most important aspects in an efficient GPU implementation. The result obtained from the GPU implementation is also dissimilar to the CPU based implementation which becomes hard to explain due to the non-deterministic atomic operations. All these shortcomings are avoided using the data driven parallelism and a more similar GPU and CPU result is obtained which is explained in Chapter 6. Though there were constraints in deciding the total number of agents and also deciding the total environment size, the data driven parallelism was preferred and better suited to the GPU.

The next subsections give a detailed description of each of the implementations attempted for modelling the crowd movement using the LEM and ACO algorithms. Initially, the modelling using the LEM algorithms was based on task based parallelism. However, due to the shortcomings of task driven parallelism, data driven parallelism was eventually adopted. Subsequently, simulation was performed for both the LEM and ACO using the data driven parallelism methodology. The following subsections discuss in detail the data driven simulation of the ACO and LEM model.

5.1 LEM based Simulation Using Task Driven Parallelism

In the LEM based simulation using task driven parallelism, a square environment was chosen for simplicity. The size is decided before the start of the simulation procedure. However, it is not mandatory for the environment to be a square. It could be rectangular and of any size. The agents are positioned on the top and bottom portion of the environment. The number of rows up to which they can occupy is user defined. The initial posi-

tions of the agents are totally random but kept confined to the number of rows decided by the user. The target or objective of the agents is to reach the other side of the environment or as far as possible into (across) the environment. For an agent placed in the bottom portion of the environment, the target is set to the reach the top row and for the agents placed in the top portion of the environment the target is set to the bottom row of the environment. Each of the agents is considered a single CUDA thread which is known as *task based parallelism*. Threads are launched to cover all the agents placed in the environment. After the total number of threads to be launched is decided, threads are arranged into blocks. The blocks are 1D and then the kernel function is launched to conduct the simulation. The environment is divided into rigid cells which is actually a coordinate based system. In a coordinate based environment, rows become the only deciding factor when an agent's only goal is to cross-over to the other side of the environment or to go as far as possible. So the column in which an agent is currently situated changes as the agent moves in the environment. However, the target row is kept fixed which is the last row of the opposite side. The target column varies according to the current position of the agent. The column in which the agent is currently located becomes the target column. Before making a move, an agent checks the availability of the forward cell. In Figure 5-1, agent M is placed in *CELL #0*, is moving from the top to the bottom of the environment. In the current position of M the target cell has the same column but is placed in the last row of the environment which is marked in blue in Figure 5-1. However, the target cell of an agent varies according to the current position of the agent. As long as the agent moves along the same column, the target cell remains constant while it changes as the agent

changes the column. An agent's target would be to reach the last row of the opposite end or to reach as far as possible.

Initially, agent M checks the availability of CELL #1. If CELL #1 is empty then no further calculation is conducted and agent moves to the forward cell. In case of the agents moving from the bottom of the environment to the top portion, the same operation would be performed on CELL #6. If the forward cell is not empty, then the agents checks the rest of the neighborhood cells which is from CELL #2 to CELL #8.

If none of the surrounding cells are empty then the agents remain in the same position as in the previous step. If the surrounding cells are empty then the LEM is carried out in the following steps. After finding the number of surrounding empty cells, they are ranked based on their Euclidian distance from the target. The cells are ranked in an ascending order where the nearest cell from the target gets the lowest rank and the farthest cells have the highest rank. While calculating the distances of the neighborhood cells from the target, distances of certain cells may be equal. It can be inferred from the picture that the distance of target from *CELL#2* and *CELL#3*, *CELL#4* and *CELL#5*, *CELL#7* and *CELL#8* would be same.

So while arranging the neighborhood cells in the ascending order of distance, one among the two conflicting cells having same distance from the target is chosen randomly. This random number is generated from the uniform distribution. Only the cells that are empty are considered and arranged accordingly. Then the arranged cells are normalized. After arranging the cells, another random number is generated using the normal distribution. A separate random number is generated for all the agents. Then the appropriate cell is cho-

sen based on the generated random number. The agent would be moving to the cell chosen in the next step.

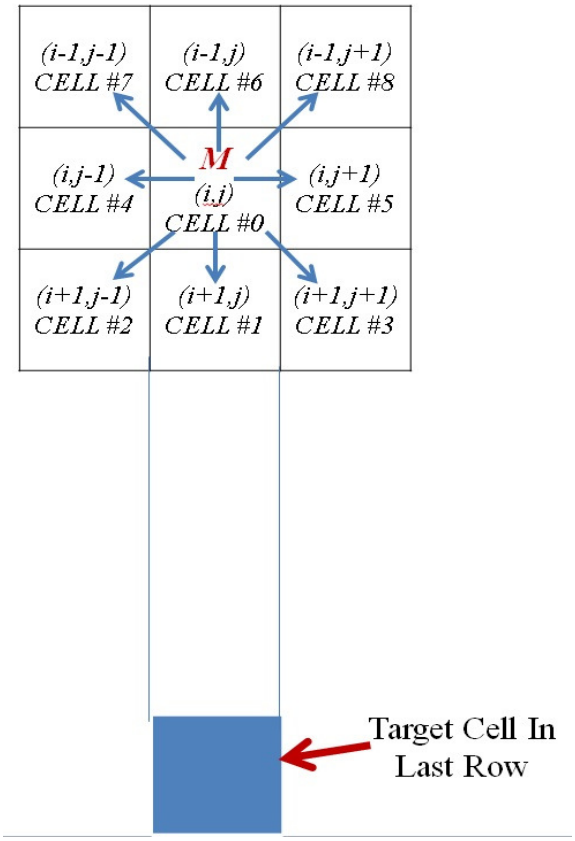


Figure 5-1: Target distance from pedestrian M placed in central Cell #0

This chosen cell is actually a memory position in the global memory of the GPU. If the thread allotted for agent M chooses a cell which is also chosen by some other agent, then two agents would be moving to the same cell on the next step. In that case, two agents would be writing on the same memory location simultaneously which would cause an error and effectively a loss of an agent as memory is overwritten. To avoid this kind of situation, some locking mechanism is required through which two threads won't be able to access the same memory location at the same time. This is achieved by using the atomic operations of the CUDA threads. In atomic operations, when a thread operates on a memory location, it actually locks the memory and forbids any other threads from operat-

ing on the same memory location simultaneously. Thus, in the pedestrian movement application when an agent tries to move, it exchanges the current with the future position using atomic exchange operations. Suppose another agent chooses the same cell and tries to move simultaneously, then while doing so, it would find it to be locked by the thread assigned to agent M. Consequently, it will not be able to write to the same memory location at that time. In this way, overwriting memory and loss of agents are prevented.

Although the atomic operations prevented the overwriting of memory locations and loss of agents, this actually serializes the program rather than making it parallel which is the price paid for using atomic operations. Using task based parallelism, there is freedom in choosing the environment size and also the number of agents; however it involves lots of control divergence which again increases serialization. In task based parallelism, the conditional statements within the kernel function might cause the threads within a warp to take different execution paths known as control divergence. This control divergence causes serialization of a parallel program and slows down the operation. This is another major disadvantage of using task based parallelism. This parallelism is suitable for applications where the number of threads is relatively small. All these issues are avoided by using data driven parallelism which is discussed in the next section.

5.2 Simulations on GPU using Data Driven Parallelism

In the task driven parallelism, there were several issues that needed to be addressed and removed to make it better suited for a GPU. These issues were addressed by converting the task based parallelism model to the data driven parallelization model. In task driven parallelism, there were lots of control divergences which were avoided in the data driven

parallelism implementation. Atomic operations were used in the task driven parallelism implementation to avoid the loss of agents (memory overwriting) during the simulation. However, using the atomic operations increases serialization. With data parallelism, some advanced techniques are used through which the same purpose is achieved without using the atomic operations. Apart from this, care was also taken towards reducing the global memory traffic by introducing a tiling technique to increase the operations on the shared memory data. The resources in the GPU vary from one generation to another which is indicated by the compute capability of the GPU. Utilizing the GPU resources properly and maintaining the occupancy of the GPU to 100% is very important. This involves maintaining the number of threads that can be launched within a block, not exceeding the amount of shared memory that could be used by a block of threads and maximising number of registers that could be used inside a thread. Optimization is also achieved in the data driven model by reducing the instructions by unwinding the loops. These issues were addressed while using the data driven parallelism. Data driven parallelism is used to model both the LEM and a modified ACO based model. The implementation is divided into separate stages each performing a particular portion of the simulation. Each of the stages is almost the same in both the LEM and ACO simulation. The next sub-sections discuss each of the stages for both simulations.

5.2.1 Phase I: Data Preparation Stage

This is the initial phase that is carried out for both the LEM and ACO based simulation on the CPU side. This stage is responsible for preparing the data that would be transferred to the GPU. This stage is carried out only once outside the main simulation loop. While

implementing this stage, there are certain constraints that are needed to be considered. These constraints are imposed because of the GPU generation being used for the simulation. A GPU with a compute capability of 2.0 is used here and maintaining the occupancy to 100% imposes the constraints. These constraints are that the environment is chosen to be of square size and the length of the sides is user defined. To maintain 100% occupancy, the maximum number of threads that would be present in a block is 256. As discussed previously, a tiling method is adopted to perform the simulation. The whole environment is divided into square tiles of equal sizes. Within the simulation, each of the CUDA threads are assigned to each of the cells in the environment and each thread block is mapped to each of the square tiles. As the maximum number of threads that could be launched is 256, each tile size is 16x16. To maintain the occupancy of the GPU, a side of considerable length (user-defined) and multiple of 16 is chosen. An environment of 480x480 total square size is chosen. The primary environment matrix is known as *mat* matrix where all the agents are placed. A sample environment is shown in Figure 5-2. The sample environment is of size 16x16 which is actually equivalent in size to each tile of the larger environment. The agents are placed in two opposite portions of the environment divided into two groups. The numbers of agents are confined to certain number of rows (e.g. Rows == 3). Both the top and bottom placed agents are indicated by arrows. The top placed agents are labeled by 1 and the bottom placed agents are marked by 2. The empty cells are marked by 0. At a point of time, each of the agents would occupy only a single cell and are free to move to any of the neighborhood cells. The sizes of all the agents and their velocity are considered to be constant throughout the simulation. Anoth-

er matrix of the same size as the *mat* matrix is created which is known as the *index matrix*.

The *index matrix* functions to counts the number of agents in the environment. It starts from 1 and then the count increases up to the last agent in the environment. The first agent is marked as 1 and then a scanning operation is performed through every column and row. When the second agent is detected, then it is marked as 2, 3 when the third agent is detected and so on until the last agent. This operation basically tags the agents in the environment with a unique number which is used to access the rows of some additional matrices inside the simulation process. A sample *index matrix* is shown in Figure 5-3 of size 16x16. Scanning operations starts from the first row and first column.

A top placed agent having label 1 is encountered in the first row and first column and it is marked as 1. As the scanning operation continues, another agent is confronted in the first row and the fourth column and it is marked as 2 in the *index matrix*. The last agent is bottom placed agent in the 16^{th} row and 16^{th} column having label 2 which is marked as 58. This is the last agent placed in the environment and so the tagging stops.

Another matrix is created which is known as the *property matrix* and it is used to maintain the property of each agents. The content of a single row of the property matrix is depicted in Figure 5-4. Table I illustrates the contents in further detail.

In each row of the *property matrix* there are 8 columns. The total number of rows in the *property matrix* is equal to the total number of agents in the environment plus one. For the sample environment shown in Figure 5-3, the total number of rows in the *property matrix* would be 59 as the total numbers of agents were 58. The index number of the

agents in the *index matrix* is used to indicate the row number in the other matrices like the *property matrix*. In this application, there are three kinds of situations that could arise.

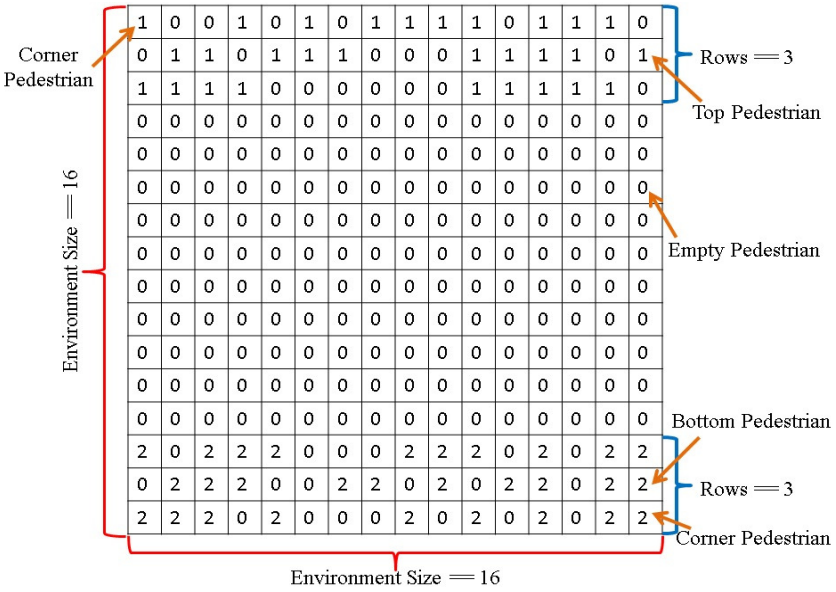


Figure 5-2: Sample original matrix *mat* of size 16x16

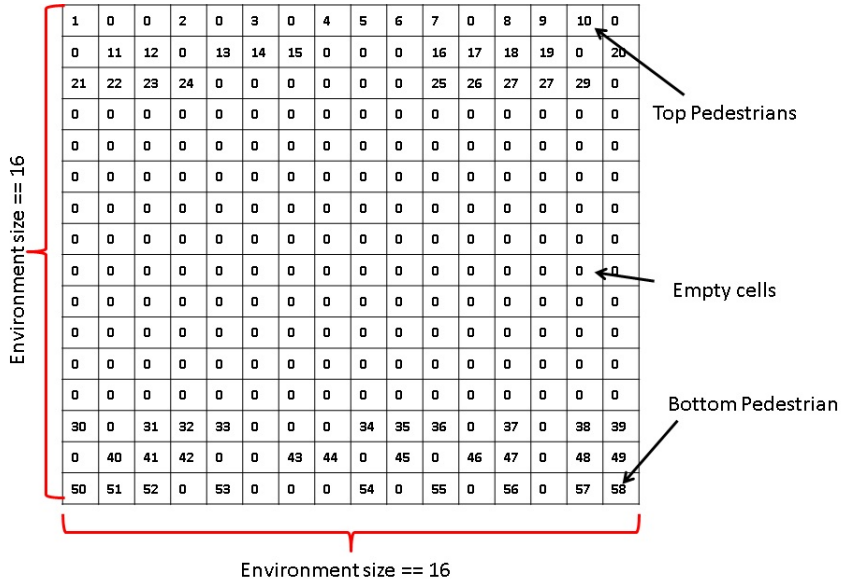


Figure 5-3: Sample *index matrix* same as *mat* of size 16x16

ID	INDEX NO.	ROW	COLUMN	EMPTY	FUTURE ROW	FUTURE COLUMN	FRONT CELL

Figure 5-4: A row of the *property matrix*

Table I. Illustration of the contents of property matrix

Column #	Name	Description
1	ID	Identity of the pedestrian, either 1 or 2
2	INDEX NO	The index value of the matrix from the index matrix.
3	ROW	The present row position of the pedestrian.
4	COLUMN	The present column position of the pedestrian.
5	EMPTY	Unused
6	FUTURE ROW	Holds the future row value that would be decided in the subsequent steps, initially 0
7	FUTURE COLUMN	Holds the future column value that would be decided in the subsequent steps, initially 0.
8	FRONT CELL	Holds the value of the value that is present in the front cell from the current position.

A cell in the environment could be occupied by a top or bottom placed agent or it could be empty. While calculating in the latter phases, the empty cells should be discarded and only the threads assigned to the occupied cells should be considered. That is why in the *index matrix* the occupied cells are marked with a unique number and the empty cells are marked as 0. The threads assigned to the occupied cells of the main *mat* matrix stores the result after calculation in the row number of the *property matrix* indicated by the same cell of the *index matrix*. The calculations performed by the threads assigned to the empty cells are stored in the 0^{th} row of the *property matrix.* This is the reason why there is one more row in the property matrix to store the results carried out by the threads on the empty cells. In this way, the warp divergences carried out by the threads on different kinds of agents and the empty cells were avoided. The *property matrix* is updated with the appropriate data in this phase. After the agents are placed in the environment, all the rows are columns are traversed and as the agents are encountered, the *property matrix* is updated with the required data. This is performed only once in this phase.

Another matrix known as the *scan matrix* is created similar to the size of the *property matrix*. But this is created on the global memory of the GPU only and then it is initialized

to 0. The *scan matrix* is used in the next phase of the simulation. For LEM based simulation, this matrix is used to store the result generated from Equation (2-3) and for ACO based simulation it is used to keep the result generated from Equation (2-4). The 0^{th} row of the *scan matrix* is also used to store the result of the operations carried out by the threads assigned to the empty cells. The rest of the rows are for the valid agents which could be accessed by the number indicated in the *index matrix*.

For ACO based simulation, another matrix known as *tour matrix* is created which has the same number of rows as the *scan* or *property matrix* but has only one column. It is used to keep the tour length value of the pedestrian. It is also created on the GPU global memory and then it is initialized to 0. It is used to retain the distances travelled by the agents and for the pheromone computation in the later phases of the simulation. The 0^{th} row is also used to store the result generated from the empty cells and the other rows are used to store the result generated by valid agents. The row number to be accessed is obtained from the *index matrix*.

One of the most important matrices in the ACO based simulation is the *pheromone matrix.* Two separate matrices are created for the top and bottom placed agents. Each of the matrices is equal to the size of the main *mat* matrix and separate matrices are used to keep the track of the pheromones generated by separate type of agents. Initially the pheromone matrix is created on the CPU side and then it is loaded to the GPU global memory.

GPUs come with a wide range of memory options with different features. One of such memory is the constant memory. The data stored in the constant memory are for read only purposes. The data inside it cannot be written inside the program. The data stored in

the constant memory is cached and can be accessed at a much faster speed than the global memory. The distances of all the cells from the target are pre-calculated and stored in the constant memory. The calculation of the distance and the storing of them in the constant memory is illustrated in Figure 5-5.

In Figure 5-5, a sample square environment is depicted having 8 rows (R_1 to R_8) and 8 columns (C_1 to C_8). Consider two top positioned agents A_1 and A_2 placed in the environment in rows R_3 and R_2 and Columns C_2 and C_6 respectively. The targets of the two agents are placed on the end row on the opposite side of the environment which is R_8. T_1 and T_2 are the target cells for the agents A_1 and A_2 respectively. For an agent placed in a particular cell of the environment, the neighboring row can have only two types of distances. For example, in case of agent A_1, R_4 is a neighboring row. So distance (D_1) of the cell of coordinate (R_4, C_1) from the target cell T_1 is same as the distance of the cell with coordinate (R_4, C_3) and so they are both marked with same colour (red).

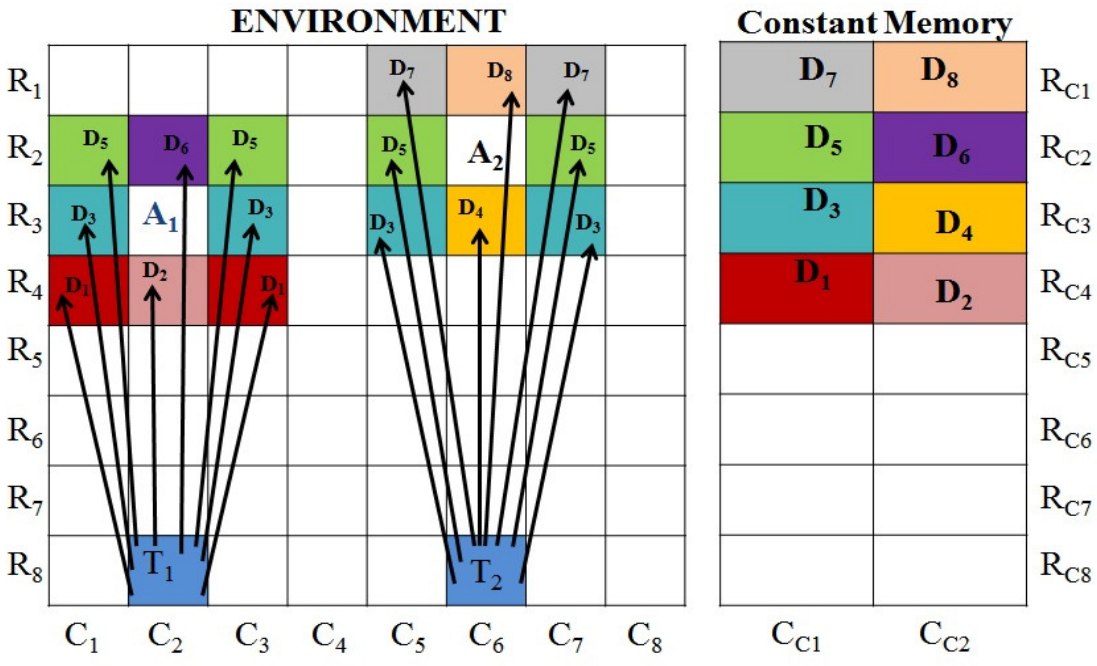


Figure 5-5: Illustration of distance calculation and storing in constant memory

However, the distance of the cell from the target with coordinate (R_4, C_2) from T_1 is D_2, which is different from D_1 and so it is marked with a different colour. Similarly for agent A_2, two neighboring cells with coordinate (R_3, C_5) and (R_3, C_7) have the same distance D_3 and are marked by the same colour, while the cell with coordinate (R_3, C_6) have a different distance D_4 and is marked by different colours. In Figure 5-5, the cells with the same distance are marked with the same color. It is apparent that based on the position of an agent, for a particular row there could be only two distinct distance values from the target cell. For a definite environment size and agent placements, the total numbers of possible distances from the target are twice the number of rows. This observation reduces considerably the constant memory requirement to store all the possible distances. Thus, in Figure 5-5, it is demonstrated that the constant memory cells with coordinates (R_{C1}, C_{C1}) store one kind of distance, whereas (R_{C1}, C_{C2}) store another kind of distance. This concept is valid for storing the distances of other cells on constant memory. The distances that constant memory cells store are marked by their corresponding colours of the environment. The figure shows the distances calculated for the agents placed on the top of the environment. Constant memory of the same size is required to store the distance values for the agents placed on the bottom portion of the environment. Accordingly, the total amount of constant memory required to store all the possible distances for both the top and bottom agents is given in Equation (5-1).

$$T_C = 2 * (2 * R_T) \tag{5-1}$$

$T_C = Total\ amount\ of\ Constant\ Memory$

$R_T = Total\ number\ of\ rows\ in\ the\ environment$

In the subsequent simulation steps, the constant memory cells are accessed by an index operation which is much faster than calculating the distances within the kernel function.

5.2.2 Phase II: Initial Calculation Phase

This phase is carried out on the GPU inside the main simulation loop. This phase is similar for both the LEM and ACO based simulation. In GPU based simulation, *tiling* is a very important concept. As explained earlier, in *tiling*, the environment is divided into tiles of equal sizes. After organizing the thread blocks, data are brought from the global memory to the shared memory of the device. Global memory is located off-chip which is slow in accessing data. In GPU based computing, it is necessary to bring the data from the off-chip memory to the faster on chip shared memory and then to operate on the local data. This reduces the latency of global memory traffic and improves the performance. Threads in the pedestrian movement application are assigned to each cell of the environment and each tile is mapped to each thread block. For the GPU used here, to maintain 100% occupancy, a maximum of 256 threads can be launched in a single thread block. The whole environment is divided into square tiles of size 16x16 and then each of the thread blocks of same size are mapped to the tiles of the environment. While bringing data from the global to shared memory, it is necessary to maintain the memory coalescing. Otherwise, a greater number of memory access instructions would be issued and a longer time would be required to access the data.

The purpose of this kernel is to determine the availability of the surrounding cells and then to obtain their distance to the target from constant memory to calculate Equation (2-3) for LEM based simulation and the numerator of Equation (2-4) for ACO based simula-

tion. In a single tile, there could be agents placed on the corner or in any other position of the tile. For the agents placed on the edges, neighborhood calculation would involve cells from an adjacent tile. Figure 5-6 shows a sample environment of size 16x16 which is divided into 4 tiles each of size 8x8. An agent is placed on the coordinate (R_1, C_8) in tile T(0,0). When this agent checks the occupancy of the neighboring cells, it has to take the cell (R_1, C_9) into consideration from the adjacent tile. Thus, to calculate the neighborhoods of all the agents in a tile, one more element is needed on all sides of the tile. For example, while calculating the neighborhood of the agents placed along the column C_8 of tile T(0,0), elements from column C_9 of tile T(1,0) are also needed. Similar cases would be observed for all the agents placed on column C_1 and row R_1 and R_8. For a tile of size 8x8, a tile of size 10x10 in the shared memory needs to be declared. An extended tile $S_T(0,0)$ of size 10x10 for tile T(0,0) of size 8x8 is shown in the Figure 5-6. Similarly for tile T(1,0), an extended tile $S_T(1,0)$ of size 10x10 is declared in the shared memory. The elements belonging to the same tile is known as the *internal elements* whereas elements loaded from the adjacent tile are known as the *halo elements*. While loading the *halo elements,* different situations could arise. The *halo elements* for the corner positioned tile are not present at all. Apart from that, loading of the halo elements into the shared memory in a traditional way involves control divergences. A different technique is used to overcome all these deficiencies.

In the original application, each tile is equivalent to a thread block of size 16x16. To carry out the necessary computation in the shared memory, a matrix of size 18x18 needs to be declared. To avoid the warp divergences while loading the *halo elements* into shared memory, a novel technique is used illustrated in Figure 5-7. The first two rows of the

thread block consist of 32 threads which makes a warp. These first two rows are used to load all the halo elements needed for that particular block. As showed in the figure, the first two rows marked in green (and one cell in red) is used to load all the halo elements on all sides marked in blue. This is done by mapping the threads using index mapping and logical operations. Through this process, all thread divergences were avoided.

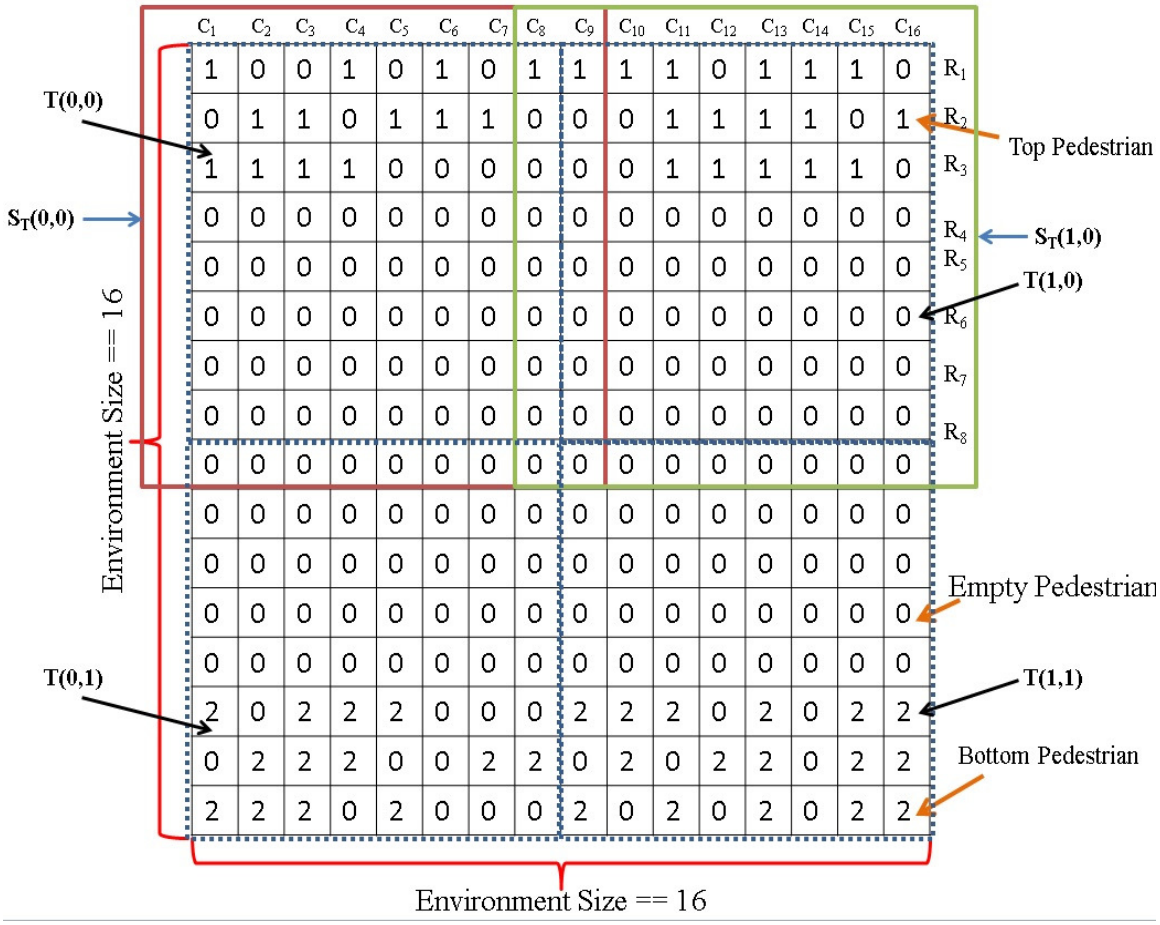


Figure 5-6: Illustration of loading halo elements

The tiles that are located on the edges of the environment have at least one of its sides where data are actually not present in the original *mat* matrix. For those cells, a dummy value is stored in its corresponding shared memory matrix. Separating the loading of these absent *halo elements* from the real ones is performed using logical operations.

In this stage for the LEM based simulation, Equation (2-3) is carried out. This particular operation is carried out by the threads that are assigned to the occupied cells and they consider only the empty neighbourhood cells while carrying out the operations. In Figure 5-1 the central CELL #0 is occupied by agent M.

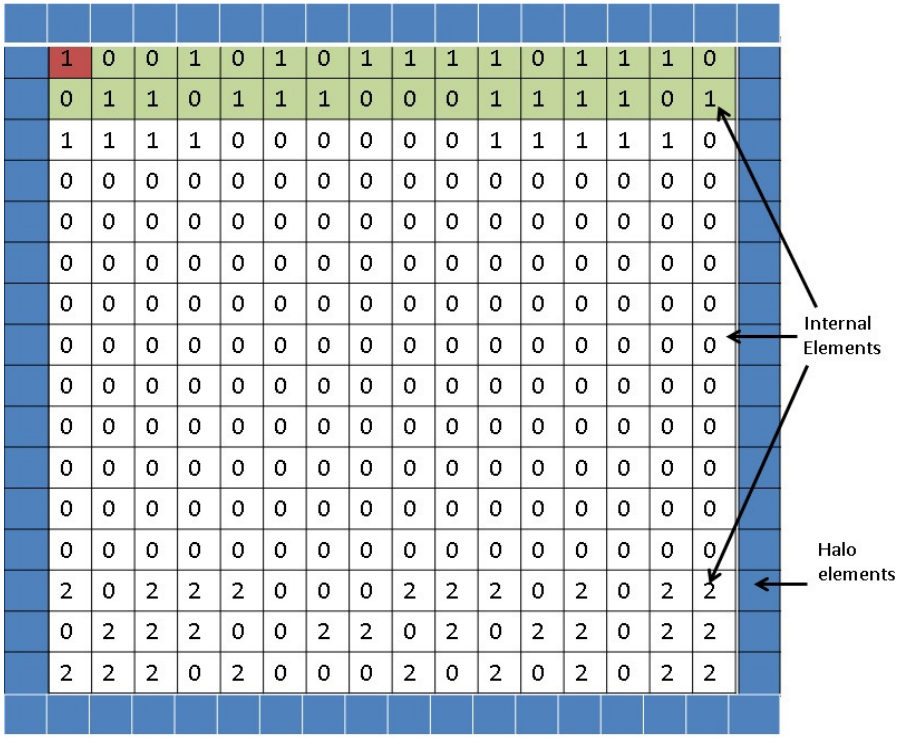


Figure 5-7: Loading of halo elements from global to shared memory

The thread assigned to this cell carries out the operation of calculating (2-3). While performing the operation, the thread would only consider the neighborhood cells that are empty. Neighborhood cells that are occupied are neglected. This calculation is accomplished using index operation and logical operators avoiding any warp divergence. Consider the agent M placed in *CELL #0* traversing from top to the bottom of the environment. In this case *CELL #1* will be nearest to the target cell. *CELL #2* and *CELL #3* have equal distance from the target which is greater than *CELL #1* but less than rest of the neighbourhood cells. The same process is carried out for the remainder of the unoccupied

adjacent cells. This whole process is carried out in reverse for the bottom placed agents. For the bottom placed agents, *CELL #6* is nearest to the target. After that *CELL #7* and *CELL #8* have the same value of distance from the target which is greater than *CELL #6* but less than other cells like *CELL #4, #5, #2, #3, #1*. In the same way, other empty cells distances will also be calculated. The required distance of the neighbourhood cells from the target is already kept in the constant memory *distance matrix*. The fetching of this distance value from the constant memory is performed using index operations. The distances fetched from the constant memory are kept in the *scan matrix* in an increasing order. The distance of the lowest value is stored first, then the next distance and so on. The row of the *scan matrix* to be accessed to store this data is obtained from the local *index matrix* residing in the shared memory. In the case of the *index matrix*, only the inner elements are loaded and so a local matrix the same size of thread block is declared in the shared memory.

For ACO based simulation, almost the same operations are carried out. However, to carry out the numerator of Equation (2-4) in the case of ACO based simulations, the pheromone matrix is needed. There are two distinct matrices for the top and bottom agents. These two separate matrices are loaded into a single local matrix in the shared memory. In case of the loading of pheromone matrices in the shared memory, the *halo elements* are also loaded along with the *internal elements*. In this way, a matrix of double the size of a local *mat* matrix is declared in the shared memory. For example within the simulation, each of the tiles are of size 16x16. In the shared memory, a single matrix of size 36x36 is declared to hold the pheromone values for both the top and the bottom placed agents. So when performing the computation for the top placed agents, the first 18 rows and 18 col-

umns are needed to be accessed and for the case of bottom placed agents, the next 18 rows are accessed. Pedestrian labels (i.e. 1 and 2) are used to access the proper portion of the matrix. By multiplying the expression by the agent label, the proper portion of the matrix can be accessed. The distance is also required to calculate the heuristic value of the numerator of (2-4). These distances are already present in the *distance matrix* of the constant memory. The distance value is obtained from the proper row using logical operation and the agent label. After obtaining all the necessary values, the numerator of Equation (2-4) is calculated and stored in the *scan matrix*. The row number to be accessed in the *scan matrix* to store the value is obtained from the local *index matrix* residing in shared memory. In this case, also the 0^{th} row of the scan matrix is used to store the values generated from the threads assigned to the empty cells. In this way, the warp divergences are avoided as well. Care is also taken towards the maximum amount of shared memory and the registers that could be used by the block and threads respectively, maintaining 100% occupancy. After this stage another kernel function is launched inside the main simulation loop which is known as the *tour construction phase*.

5.2.3 Phase III: Tour Construction Phase

This phase is carried out within a separate *kernel function* within the main simulation loop. This phase is similar for both the ACO and LEM based simulation model with some subtle differences. In this phase, agents choose the coordinate of their next position in the environment and update it in their property matrix. The row and column of the next position is kept in the FUTURE ROW and FUTURE COLUMN of the *property matrix*. The phase starts by dividing the *property matrix* into tiles and then loading them into the

shared memory. As already mentioned, the total number of threads that could be launched is 256. Each of the threads would be assigned to each of the cells of the *property matrix* and there are 8 columns in the matrix. Thus, a maximum of 32 rows could be present in a block. In this way, the maximum number of threads would be 256 (32x8), maintaining the 100 % occupancy. This is the reason behind choosing the total number of agents to be a multiple of 32. While loading the *property matrix* from the global to the local *shared memory,* the 0^{th} row is ignored as it contains the invalid data generated by the empty cells from the previous phase. Data from the 1^{st} row are loaded to the shared memory as those data were generated by valid agents.

For the LEM based simulation, each row of the local *scan matrix* contains the distances of the surrounding empty cells from the target in an ascending order. In Figure 5-8, a single row of the *scan matrix* is depicted which stores the distance of the neighborhood cell for a particular agent.

D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8

$$D_i = Distance\ of\ CELL\#i\ from\ the\ target$$

Figure 5-8: Distances of the neighborhood cells from the target in a single row of *scan matrix*

If this is a top placed agent, then D_1 is the distance of *CELL #1*, D_2 and D_3 contains the distance of *CELL #2* and *CELL #3* and it continues like this up to D_7 and D_8 which contains the distance of *CELL #7* and *CELL #8.* For bottom placed agents, it would contain the distance of *CELL #6* in D_1, *CELL #7* and *CELL #8* in D_2 and D_3 respectively and ultimately *CELL #2* and *CELL #3* stored in D_7 and D_8.

As mentioned, each block in this phase has 32 rows and 8 columns. These 32 rows belong to 32 agents. So in this kernel, the first 4 rows of each block consist of 32 threads

which make a warp. These 32 threads are mapped to the 32 rows and then a reduction operation is performed.

For normalization purposes, all the eight columns of each row are added by mapping the threads from the first 4 rows of the block. In Figure 5-9, the *scan matrix* is shown with 8 columns (C_1 to C_8) and 32 rows (R_1 to R_{32}). The first four rows in Figure 5-9 are coloured, as the threads assigned for those rows are involved in the reduction operation. The thread assigned for the cell with coordinate (R_1, C_1) operates on the reduction operation of row R_1. Similarly, the thread assigned for the cell with coordinate (R_1, C_2) is mapped to the row R_2 and its reduction.

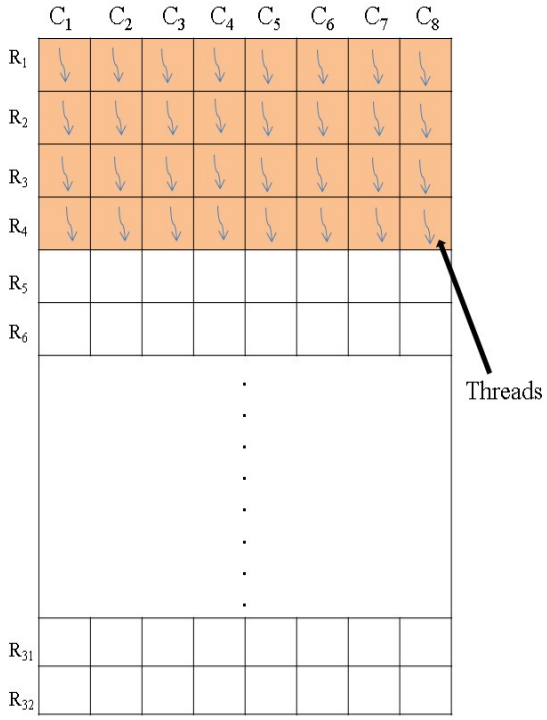


Figure 5-9: Illustration of reduction of the *scan matrix*

The threads assigned for the whole first row carries out the reduction of the first 8 rows of the *scan matrix*. The threads assigned for the second row of the matrix is mapped for the reduction of the rows from R_9 to R_{16}. In this way, the threads of the fourth row R_4 are

assigned to the last 8 rows of the local scan matrix tile. As the four rows consist of 32 threads, the warp divergences were avoided by using the technique mentioned above. While performing the reduction mechanism, instruction optimization is improved by using a loop unrolling technique. In automatic loops, like while and for, there are some additional instructions involved like checking the boundary conditions. If the loop is not big enough then it can be unrolled to avoid the additional instructions. All the 8 addition operations in the reduction operation are carried out without using an automatic loop operation. After performing the addition operations, the value in column C_8 contains the sum of all the cells of that row which is then used for normalization.

After the normalization process is over, a random number is generated using the CURAND library. The number is generated from the normal distribution with mean of 0. The random number generated contains negative values and values greater than 1. These values are rounded to bring them into valid range. This checking of the values and converting them into the valid range are performed by using logical operations avoiding all the warp divergences. As mentioned previously, the main priority of agents in the bi-directional movement is to move forward as much as possible. So the forward cell has the highest priority as it is nearest to the target. For a top placed agent placed in *CELL #0*, the forward cell is *CELL #1* and it would be *CELL #6* for a bottom placed agent. So if the forward cell is found empty, then the agent decides to move to the forward cell without making any further calculation. However, if the forward cell is found occupied, then the agent chooses the cell indicated by the random number generator. The coordinate of the chosen cell are stored inside the FUTURE ROW and FUTURE COLUMN of the *property matrix*.

For ACO based simulation, the basic steps remain the same as the LEM based simulation. The *scan matrix* is again divided into tiles each of size 32x8 and then it is loaded into the local shared matrix. The 0^{th} row is again ignored as it stores all the result generated from the empty cells. Each thread is assigned to each cell of the local *scan matrix* keeping the total number of threads to 256 maintaining 100% occupancy. In this simulation, the reduction operation of addition is the same as the LEM based simulation. The reduction operation is carried out by the threads assigned to the first 4 rows of the *scan matrix*. The loop unrolling technique is used to perform the reduction operation. After carrying out the addition operation, the cell in the last column (C_8) for all rows contains the summation result of all the columns in the same row. The value stored in C_8 is used for dividing the values in the other columns of the row with the last column (C_8). A random number is generated using CURAND library without any rounding. This random number determines the cell to be selected. The front cell has the highest preference in this simulation also. If the forward cell is empty then no further calculation is performed. Otherwise another cell is chosen by using the generated random number. After the cell is fixed, its coordinates are stored in the FUTURE ROW and FUTURE COLUMN in the *property matrix*.

After this stage is over, the next phase is launched in a separate kernel. In the next phase the agent actually moves to the next cell and the values are updated accordingly.

5.2.4 Phase IV: Agent Movement Phase

This phase is carried out in the original simulation loop in a separate kernel. In this phase, the movement of the agents takes place and matrices are updated. This phase is also very

similar for both the LEM and ACO based simulation. The block and grid configuration that is launched in this phase is similar to the *initial calculation phase*. The whole grid is equivalent to the original environment *mat* matrix. Each of the blocks needs to have 256 threads to maintain thread occupancy. The environment is broken into tiles of the same size as the *initial calculation phase* (16x16) and the thread block of the same size are mapped to each tile. In this phase, tiles of original *mat* matrix are loaded into the shared memory. The *halo elements* along with the *internal elements* also need to be loaded in the local shared matrix. So a matrix of size 18x18 is declared in the local shared memory to load the *halo elements* along with the *internal elements*. The matrix is loaded in a process as illustrated in Figure 5-7. The *internal elements* are loaded by all the threads of the block and the *halo elements* are loaded by using the two upper rows of the matrix. The *index matrix* is divided into tiles and loaded in the local shared memory. The *halo elements* of the *index matrix* are needed along with the *inner elements* in this phase of the simulation. However, in this phase the *pheromone matrix* for both the top and bottom placed agents does not require *halo elements* and only the *inner elements* are loaded in the matrix declared in the shared memory. Two separate *pheromone matrices* for the top and bottom placed agents are transferred into a single pheromone matrix declared in the shared memory. In Figure 5-10, on the right hand side is the whole pheromone matrix. The thread block shown on the left hand side of the Figure 5-10 is of size 16x16. The pheromone matrices for both top and bottom placed agents are also divided in tiles of size of 16x16. Only the inner elements of the *pheromone matrix* are necessary in this phase. So this single matrix is shown on the right hand side of the image with 32 rows and 16 columns. The top half of the local matrix with 16 rows (R_1 to R_{16}) and 16 columns (C_1 to

C_{16}) is coloured green and is used to store the tile of *pheromone matrix* for top placed agents. The bottom 16 rows (R_{17} to R_{32}) and 16 columns (C_1 to C_{16}) is the same for bottom placed agents and coloured blue in the figure.

The same thread block is used to access both portions of the matrix. An agent label is used to access the proper portion of the local *pheromone matrix*. For top placed agents, the thread block uses agent label 1 to access the top portion which is indicated by the red arrows. The yellow arrows indicates the mapping of the thread block to access the bottom half of the local *pheromone matrix*. The bottom placed agent's label is used to access the lower 16 rows of the matrix as shown in Figure 5-10.

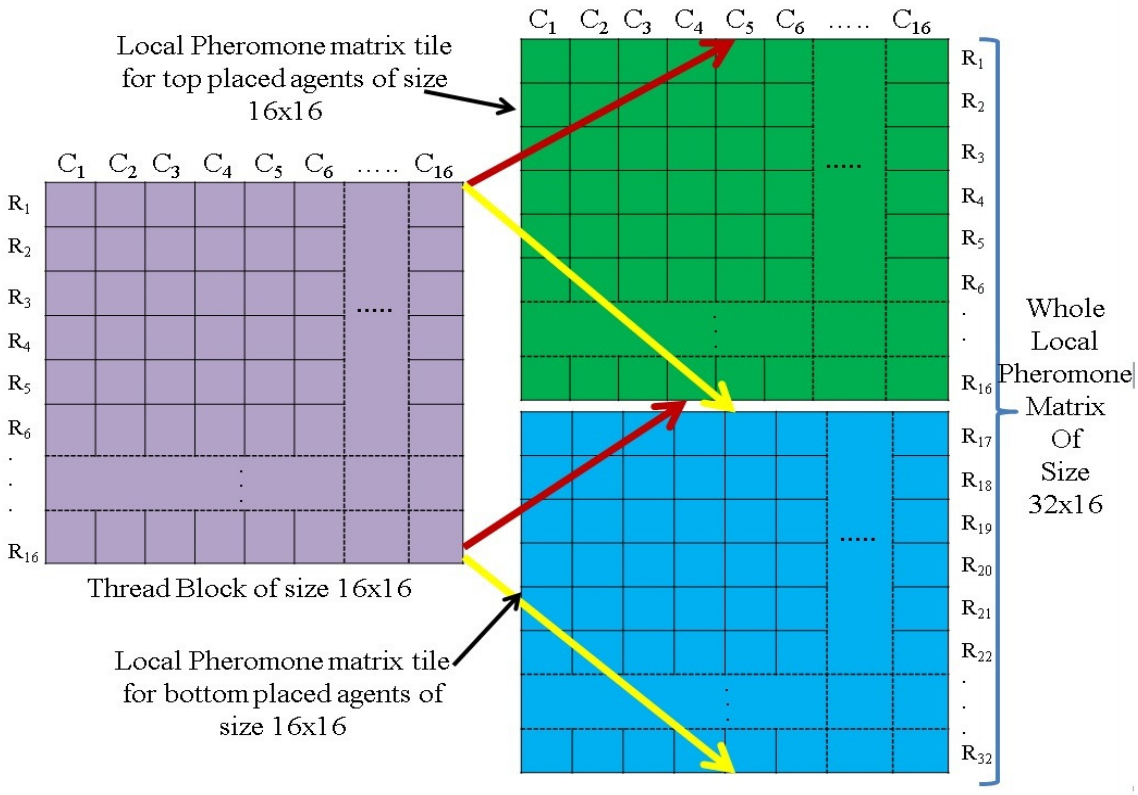


Figure 5-10: Illustration of loading data from global to local pheromone matrix and accessing of threads.

After breaking the matrices into tiles and loading the local matrices, the calculation of the agent movement begins. Contrary to the initial calculation phase, only the threads as-

signed for the empty cells participate in the calculation. The threads for the occupied cells are ignored in this phase. The reason is explained later in this section. In Figure 5-11 a typical situation is demonstrated. The cell identifiers are marked in red and the surrounding pedestrians are marked in black. The central cell with the coordinate *(i,j)* is empty. The thread assigned to this particular cell checks the surrounding cells and finds out the number of agents that are trying to move to the central cell. In Figure 5-11 there are 5 agents placed in cell 0,1,3,5 and 6 that are trying to move to the central cell. This kind of situation could give rise to race conditions. As all the threads operate in parallel, all the neighborhood agents of the central cell would try to move simultaneously and so in the end of each simulation step, loss of agents was observed. This kind of situation can be avoided in GPU programming by using the *atomic operations*.

However, atomic operations lock the particular memory location and when one thread operates on that memory location and then other threads are unable to access that location. This causes serialization of an implementation, giving rise to increase in the execution time. So to avoid both serialization and race conditions, a technique known as the *scatter-to-gather* transformation [55] is used.

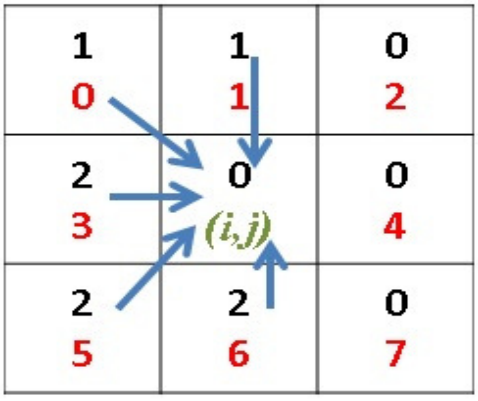


Figure 5-11: Illustration of the occurrence of a race condition

There are applications where the input elements contribute to the computation of the output element. In Figure 5-12, the scatter operation is depicted where the threads are assigned to the input cells. The threads assigned for the input cells I_1, I_2, I_3, I_4 and I_5 contribute in the operation of the output cell O_2 and threads I_3, I_4, I_5, I_6 and I_7 contribute in the output of cell O_5. All the threads that are involved operate in parallel. So while calculating the output cell O_2, there would be a conflict while updating the information. A same kind of conflict would be observed in the calculation of the cell O_5. The above operation is known as the *scatter operation* and this sort of race condition can be avoided using CUDA atomic operations.

As previously mentioned, use of atomic operation serializes an application by locking the memory cell. There is a way through which the race condition could be avoided without using atomic operations. In the next method, the threads are launched on the output cells as shown in Figure 5-13. In this method, the threads assigned to the output cells gather the information from the input cells that contribute in the calculation of that particular output cell.

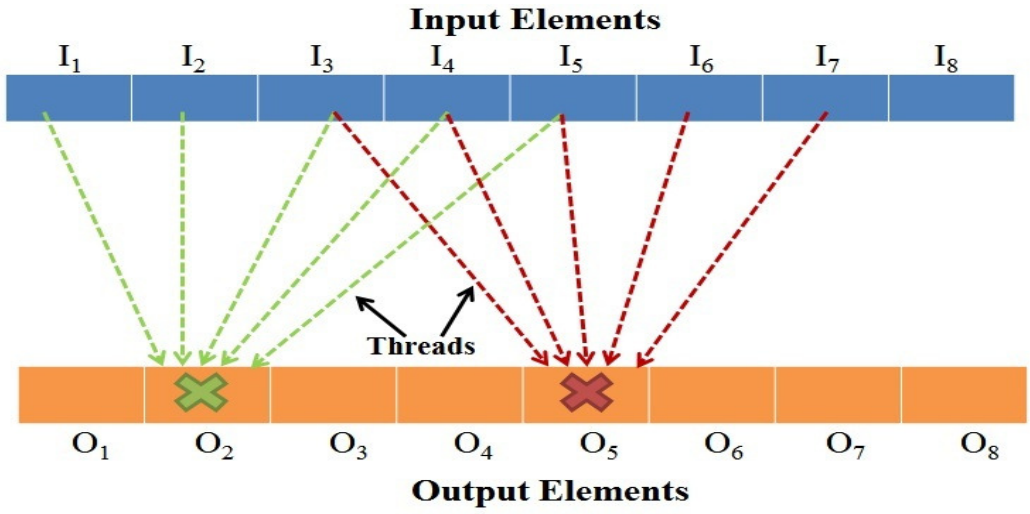


Figure 5-12: Illustration of scatter operation

The thread assigned for the output cell O_2 gather the information from the input cell I_1, I_2, I_3, I_4 and I_5 and O_5 gather the information from the cells I_3, I_4, I_5, I_6 and I_7. This prevents the race conditions and also the usage of the atomic operations. This process is known as the *gather operation* and this whole process is known as the *scatter-to-gather transformation.*

In the simulations conducted, the threads assigned to the empty cells actually represent the output and the occupied cells represent the input cells. The threads are assigned to all the cells of the environment. However, only the threads that are assigned to the empty cells enter in the computation and the threads assigned to the occupied cells are ignored..

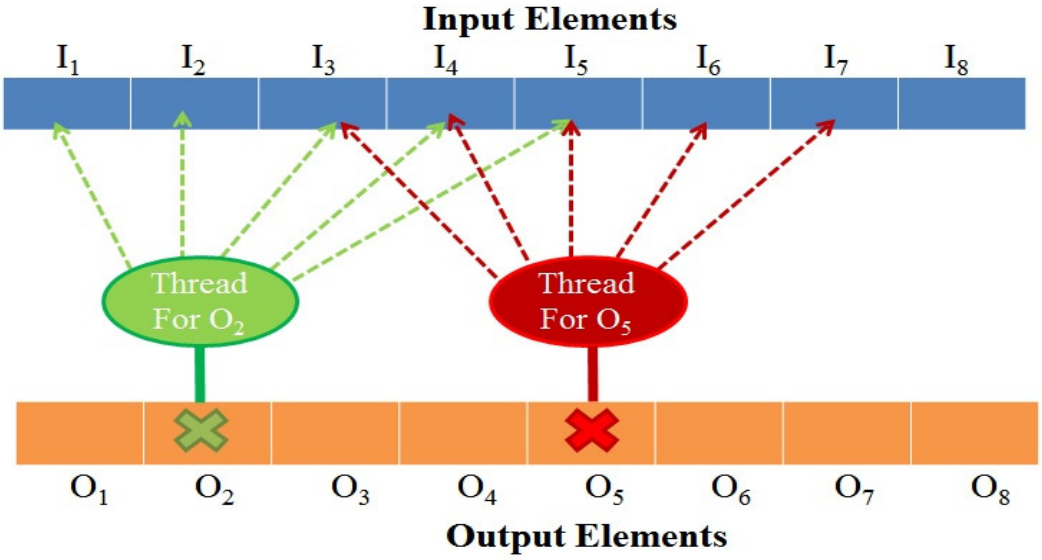


Figure 5-13: Illustration of gather operation

For example, the threads assigned for the cell 0,1,3,5 and 6 in Figure 5-11 do not involve in any sort of computation. However, the threads of the empty cells like 2, 4 and 7 are involved in the computation. The threads assigned to the empty cells scan their neighbourhood and count the number of agents that decide to move to the central cell in the next step. The information of the agent's next movement is kept in the FUTURE ROW and FUTURE COLUMN of the *property matrix*.

While performing the computation, there are some additional situations that could arise for the empty cells based on their neighborhood. There could be a situation where an empty cell *C* is surrounded by agents but none of them decided to move to *C* in the next simulation step. There could be an additional situation where none of the surrounding cells are occupied. In these situations, the result after counting the neighboring cells should be 0. So in this phase, the threads assigned to the empty cells surrounded by agents that would move to the cell in the future steps enter into the computation. The threads assigned to the occupied cells and the cells for which the counting result is 0 are ignored. The threads generating valid results are isolated from all the other threads using logical operators avoiding warp divergences. So through this process, only the cells generating a count greater than 0 are considered and the rest are ignored.

After this step, a random number is generated from the uniform distribution using the CURAND library. A random number is generated for an empty cell with a valid count. The random number is generated within 1 and the maximum number obtained after counting the number of agents moving to that particular cell. For all the other cases, the random number is converted to 0. The random number indicates the cell to be chosen to exchange its position with the central empty cell. After making the change in the original matrix, the ROW and COLUMN section in the *property matrix* is also updated. The row of the *property matrix* that needed to be accessed is obtained from the *index matrix*.

For ACO based simulation, the pheromone matrix also needs to be updated. The pheromone value of the central cell needs to be increased. However, before incrementing the pheromone values, the tour length needs to be increased in the *tour length* matrix. The tour length is unique for a particular agent. The cell in the *index matrix* corresponding to

the cell in the main *mat* matrix where the chosen agent is placed contains the row number that needs to be accessed for that particular agent. The number in the *index matrix* is used to access the row in the *tour length* matrix for that particular agent moving to the empty cell. The value that is needed to be added in the tour length matrix is procured from the *distance matrix* kept in the constant memory. The data in the constant memory are cached and so it is faster to access rather than calculating the distance from the target which involves compute intensive operations such as square roots. After the operation on the tour matrix, the pheromone value is changed in the local *pheromone matrix*. But the local *pheromone matrix* contains both the top and bottom placed agent's pheromones as shown in Figure 5-10. The proper portion of the matrix needs to be accessed for a valid operation. This is again done by using the agent's label, in that the label of the agent getting replaced is used to access the proper portion of the matrix. If the agent getting exchanged is placed on the top portion, then agent's label (1) is used to access the top portion of the local *pheromone matrix*. Otherwise if the agent is bottom placed then the agent's label (2) is used to access the bottom section of the *pheromone matrix*.

The *index matrix* gets updated after all the changes are completed in both the LEM and ACO based simulation. This is performed in the same way as performed in the *mat* matrix. Through the processes mentioned above, all the updating is performed without involving any atomic operations in both of the LEM and ACO simulations. This phase is the final operational phase after which only a supporting phase is left where some additional matrices get updated.

5.2.5 Phase V: Supporting Kernel Phase

This phase is the carried out in a separate kernel function inside the main simulation loop. In this phase, additional matrices get updated before the control again returns to the first phase of the simulation. The structure of thread block and grid launched in this phase is similar to the *tour construction phase*. However, this phase is very straightforward and no control divergences are involved. Every thread can participate in the computation and updating of the matrices. In this phase, for both the LEM and ACO based simulation the *scan matrix* gets initialized so to avoid any conflict in the future. The FUTURE ROW and FUTURE COLUMN of the *property matrix* are also initialized in this phase. After these updates, control is returned to the *Initial Calculation Phase* and the simulation is carried out again until a predefined number of iterations is reached.

CHAPTER 6

In the crowd simulation undertaken here, two algorithms, the LEM and ACO were implemented on both a CPU and a GPU. This chapter discusses performance gains obtained using the GPU. The execution time of the LEM and ACO based simulation on GPU are compared. The execution time for GPU implementation of ACO based simulation with its single threaded CPU counterpart is also compared. For the GPU based simulation, the NVIDIA GTX 560ti with FERMI architecture was used. On the CPU side, an Intel Core i7-930 processor was used. The details of the hardware specification of both the processors used are provided in Table II.

Table II. Hardware specifications of CPU and GPU used

Attributes	**CPU and GPU hardware specifications**	
	CPU	***GPU***
Manufacturer	Intel	NVIDIA
Model	Core i7-930	GeForce GTX 560ti
Processor Cores	4	448
Clock Frequency (GHz)	2.8	1.464
L1 Cache size	32 KB + 32 KB	16 KB + 48 KB (shared memory configurable)
L2 Cache size	256 KB/ core	768 KB
L3 Cache size	8 MB	Not available
DRAM Memory	6 GB DDR3	1.25 GB GDDR5

The programming environment is Microsoft Windows 7 using Microsoft Visual Studio 2010 for both the GPU and the CPU based simulation. The NVIDIA CUDA compilation

tools release 5.0 was used to program the GPU. To measure both the ACO and LEM based simulation on the GPU, *cudaevent* functions were used. On the CPU side, *time* functions are used to measure the performance of the simulations.

6.1 Performance Analysis

Through the performance analysis, the advantages behind using multi-threaded GPU for simulation purposes rather than the traditional single threaded CPU would be demonstrated. In the entire simulation process, the environment was chosen to be square of size 480x480 and kept constant in all the simulations performed. One of the constraints in the implementation is to keep the total number of agents a multiple of 32. The simulation starts with 2560 agents divided equally in each side of the environment, i.e. 1280 agents on each side. The agent's initial position is random but kept constrained within a certain number of rows. The simulation is carried out with 2560 agents for a predefined value of 25,000 numbers of steps. Then the numbers of agents are increased to 2560 agents on each side of the environment or 5120 in total. This sequence is carried out (incrementing the number of agents on each side by 2560) until the total number of agents reaches 102,400 in the environment, for a total of 40 simulations. Each of the 40 simulations is carried 10 times and their time of execution is recorded in text files and the simulation time is averaged. All simulations are carried out for a predefined number of time steps of 25000. The time measurement is performed for the data driven LEM and ACO based simulation on both the GPU and CPU. The result of the performance analysis is provided in the next section.

6.1.1 Execution time comparison of ACO and LEM simulation on the GPU

The execution time of the simulation is recorded in a file using C file operations carried out within the simulation process. Matlab is then used to process the data and produce the graphs. The graph shown in Figure 6-1 shows the execution time comparison between the data driven ACO and LEM based simulation, with each simulation performed 10 times and results averaged. In the graph, the X-axis denotes the total number of agents that are present in the environment. Recording of the execution time for both kind of simulation begins with 2560 total number of agents. Then the number of agents increases at the rate of 2560 agents on each side of the environment until the total number of agents reaches 102,400 on both sides. The Y-axis of the figure shows the execution time of the simulation in seconds.

In Figure 6-1, the execution time of the ACO based simulation is marked in green and LEM based simulation is marked in blue. The execution time of both the simulations is almost the same with a minor difference. The execution time of ACO based simulation is marginally greater than the LEM based simulation due to some additional instructions that need to be carried out. The simulation starts with 2560 agents on the environment which is shown in the X-axis. The execution time in case of the LEM based simulation is 36.15 seconds and the execution time for the ACO based simulation it is 46.66 seconds, which is approximately 10 seconds or 29% greater at 2560 agents. The total execution time increases gradually as the number of agents increases due to a larger number of agents in the simulation. However, the difference of execution time between ACO and LEM based simulation remains almost constant of 10 seconds, regardless of the number of agents in the environment. When the total number of agents reaches to 102,400, the

execution time of LEM based simulation is 116.4 seconds and that for the ACO based simulation the execution time is 126.7 seconds. There is a marginal increase of 11% in the execution time of ACO compared with the LEM based model because of the additional operations.

Figure 6-1: Execution time comparison for ACO and LEM simulation in seconds

6.1.2 CPU and GPU execution time comparison of ACO based simulation

This section provides a comparison of the execution time for ACO based simulation on the CPU and the GPU. In Figure 6-2, the graph depicts the execution time of the ACO based simulation on CPU and GPU. In this graph, the X-axis denotes the number of agents present in the environment. The simulations start with 2560 agents in the environment and increases at the rate of 2560 agents for the next simulation. The increment continues until the total number of agents reaches 102,400. The final simulation time step of 25,000 also remains constant throughout the simulation. Each simulation is carried out

10 times and execution times averaged. Initially with 2560 agents in the environment, the execution time of the simulation on GPU is 46.66 seconds, whereas on CPU the execution time is 837.5 seconds. There is a steady increase of execution time in case of both the CPU and GPU based simulation as the number of agents increases in the environment. When the number of agents reaches to 102,400, the execution time of the GPU based ACO simulation is 126.7 seconds and its CPU counterpart is 1449 seconds. In Figure 6-3, there is speed-up graph which shows the gain in speed of the GPU over the CPU implementations. The Y-axis here depicts the speedup factor which is basically the CPU execution time over the GPU execution time. Initially there is a speedup factor of 18X when the number of agents is 2560 in the environment which decreases to a speedup factor of 11x as the number of agents increases to 102,400. Figure 6-2 shows a steady increase in computation time for both the CPU and GPU based simulations and a corresponding decrease of speedup at the same time. In between the minimum and maximum number of agents considered, the decrease in speedup is fairly linear with the increase in the number of agents (Figure 6-3).

The results indicate that on the same platform, the ACO based simulation resulted in a higher execution time than the LEM based simulation. The difference in execution time was observed to be constant regardless of agent population in the environment. Between the two platforms considered – CPU and GPU – the implementation on CPU resulted in a significantly higher execution time than implementation on a GPU, although the speedup factor was diminished as the agent population was increased.

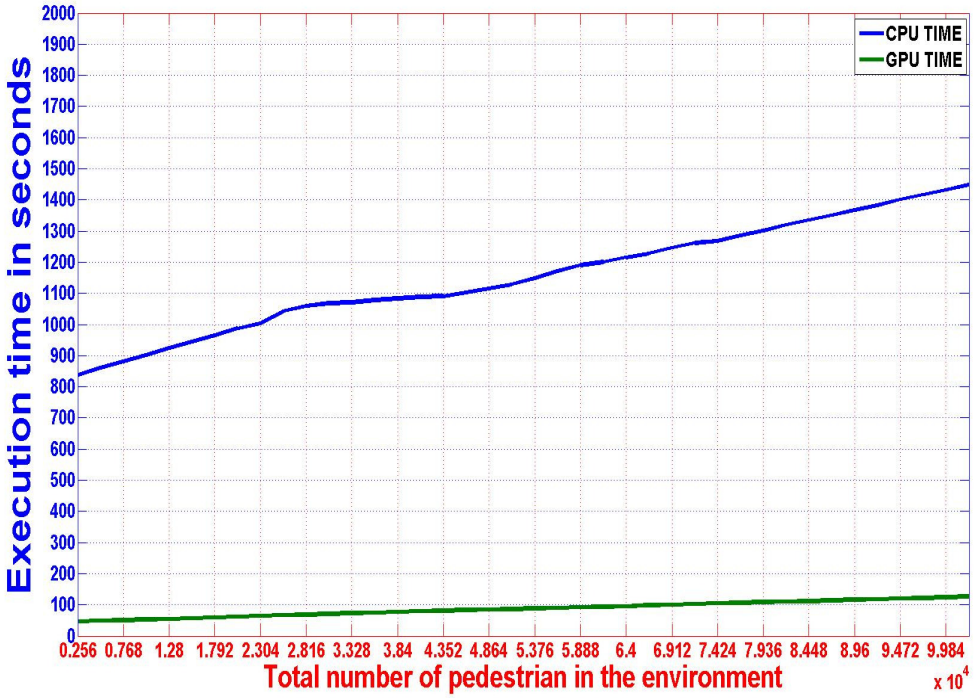


Figure 6-2: Execution time comparison of CPU and GPU using ACO based simulation

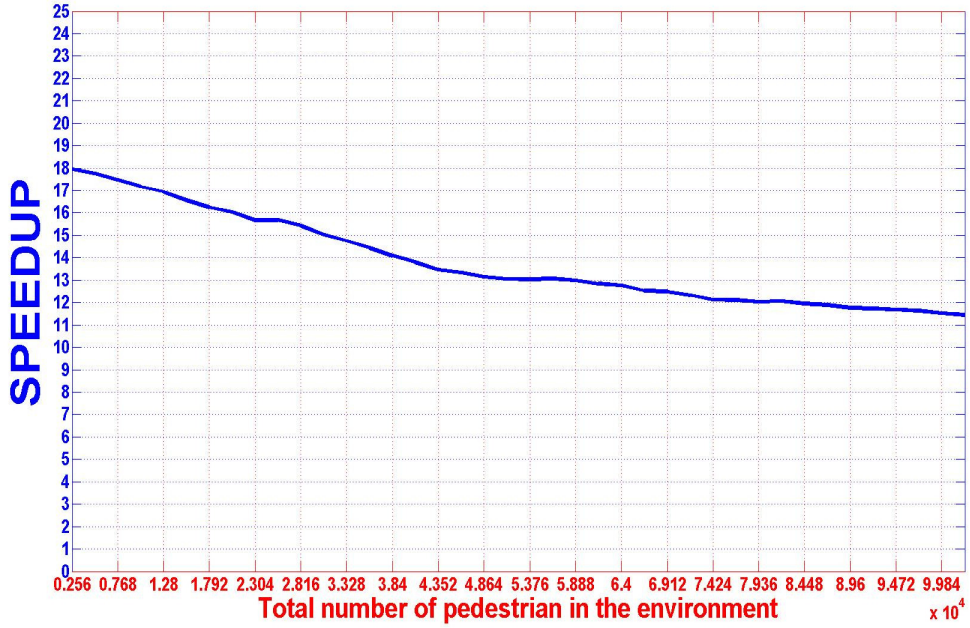


Figure 6-3: Speedup graph comparing CPU and GPU based ACO simulation

CHAPTER 7

This chapter assesses the output obtained from both the ACO and LEM simulations. The models are compared based on their output result. Through this comparison, the improvement obtained using ACO based simulation as compared with the LEM based simulation was established. A comparison between the output obtained from the CPU and GPU based simulation for ACO based simulation is also performed.

7.1 Result Analysis of LEM and ACO

This section provides a comparison of the results obtained from the LEM based simulation and its ACO counterpart, both implemented on the GPU, is provided. The definition of *throughput of pedestrians* is the number of pedestrians able to cross the environment and reach the other side in a given number of time steps. As mentioned previously, initial pedestrian positions, though random, are restricted to a certain number of rows (e.g. Rows==3 in Figure 5-2). The throughput is recorded as the number of pedestrians that are able to cross over to the opposite side of the environment. An agent is considered to reach in the opposite side when the difference between the last row on the opposite side and their present row is equal to the row number they were initially constricted to. As shown in Figure 5-2, the agents placed along the top are initially confined to the 3^{rd} row and the agents placed along the bottom are confined to the 14^{th} row. So, the agents placed along

the top are considered to reach the other side when they cross the 14th row on the opposite end and the agents placed along the bottom cross the 3rd row. The results obtained from all the simulations for LEM and ACO based simulation are shown in Figure 7-1 for the first 20 simulations, to the point where the pedestrian (agent) total reaches 51,200. Figure 7-1, shows the throughput of the agents in the ACO and LEM based simulation, both implemented on a GPU. The Y-axis is the total number of agents that are able to cross to the other side of the environment, averaged over 10 simulation runs. The X-axis represents the simulation number (agent density). At an agent population greater than 51,200, the pedestrians are either unable to cross to the other side within the fixed amount of time steps or the number that are able to cross is insignificant (total gridlock). Gridlock is used here to refer to the situation where congestion impedes any additional progress. In the models here, gridlock is a combination of both deadlock as well as livelock. From Figure 7-1 it is evident that for the first 9 simulation scenarios, the throughput for both the ACO and LEM based simulation is effectively the same. However, in the 10th simulation run where there are a total of 25,600 agents in the environment, a decrease in throughput for the LEM based model compared with the ACO based model, both simulated on a GPU, was observed. In this situation, 17,417 agents are able to cross in 25,000 time steps, whereas for ACO based model, 25,600 agents are able to cross in the same number of time steps. Maximum throughput for the ACO driven simulation is observed in the 11th simulation run with a total number of 28,160 pedestrians in the environment. For the ACO and LEM, the agent throughputs were 28,160 and 5,272 respectively. In simulation runs with agent populations greater than 25,600 (10th simulation in Figure 7-1), the throughput of ACO based model is significantly higher than the LEM based model. For

the given parameters of this work including environment size, environment configuration, and agent rules, the population size of 25,600 (10^{th} simulation) is a point of significant divergence in the performance of the two algorithms.

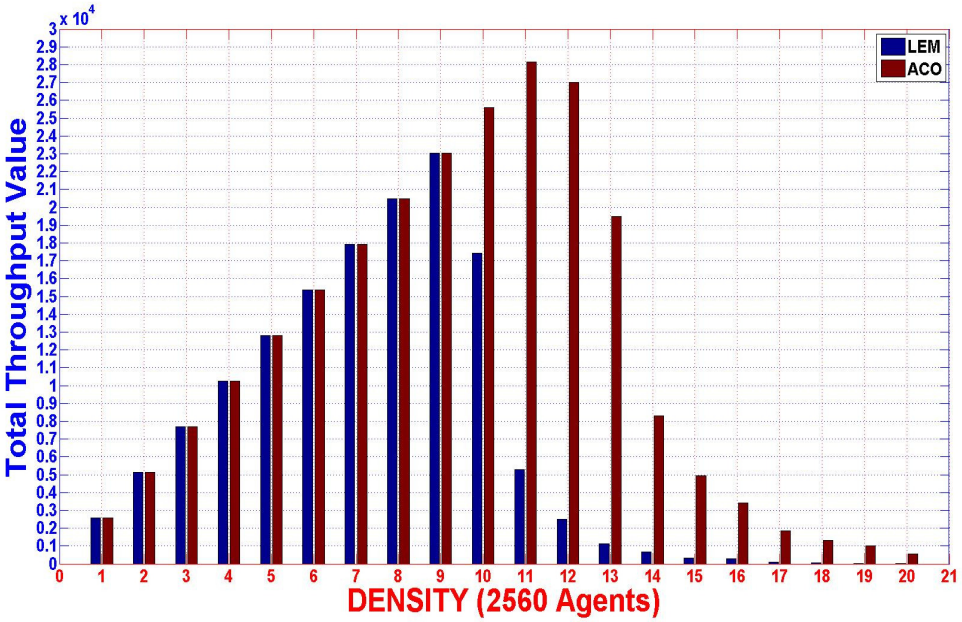


Figure 7-1: Graph comparing the throughput of pedestrians for LEM and ACO model implemented on a GPU

Considering all the 20 simulation scenarios, there is an overall increase of 39.6% in the throughput using the ACO based model over that of the LEM based model. The higher throughput obtained using the ACO-driven simulation is interpreted as a better result than the LEM-based simulation as a consequence of the pedestrians having greater agency or decision making capacity. In the next subsection a comparison between the ACO simulations conducted on the CPU and GPU which strengthens confidence in the validity of the implementation method that has been adopted. The ACO model was used as the basis of comparison as it is considered a better model of pedestrian movement for this particular instance or environment.

7.2 CPU and GPU Output Comparison

In this section, the output obtained from the ACO based simulation for both the GPU and CPU is discussed. The quality of the simulation results obtained from the GPU implementation is assessed by comparing it with the CPU counterpart. In the simulations performed, ACO benchmarking is not possible as was done in [50] using TSPLIB [51] as the problems are dissimilar. Comparing the solution obtained from CPU and GPU however is a viable way to begin to establish consistency of the implementation, as part of an overall validation of the approach. In this case, the process is similar to model docking. “Docking, also known as `alignment' or `replication', is a form of V&V that tries to align multiple models in order to investigate whether they yield similar results.” [52]

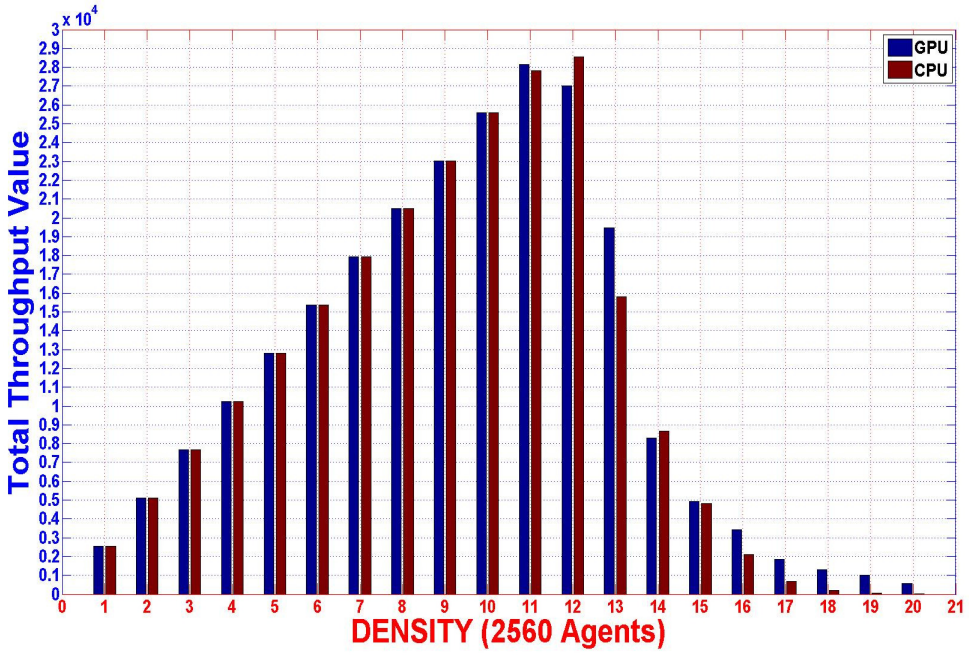


Figure 7-2: Comparison of throughput on CPU and GPU for ACO based pedestrian simulation

Figure 7-2 depicts the comparison between the throughputs of ACO based simulation on the GPU and CPU. The results are obtained after carrying out each simulation scenario 10 times and then averaging the results. The result shown in Figure 7-2 is again recorded

for the first 20 simulations when the total number of agents is 51,200. Beyond that number of agents, the throughput is insignificant due to total gridlock observed in the environment after the number of agent reaches 51,200. The throughputs for both the CPU and GPU based simulation are almost identical. Subtle differences are observed from the 11th simulation run onwards. However, apart from the 12^{th} simulation run (with total of 30,720 agents) the throughput for the GPU based simulation is comparable to the CPU based simulation. To compare GPU performance with CPU performance binomial generalized linear model (glm) [53] is used, where the probability that an agent crosses over to the other side is modeled with respect to the different number of agents and an indicator for the simulation run being run on either the CPU or GPU. There are 40 different simulation scenarios each with variable agent sizes (1^{st} scenario starting with 2560 agents and 102,400 agents in the 40th scenario). The first 10 scenarios and the last 10 scenarios are suppressed for comparing CPU and GPU. For most of the first few scenarios, all agents cross over in both cases, while in most cases in the last 10 scenarios none of them actually cross over. The test for statistical significance of the difference of performance between CPU and GPU in this model used a t-test, and did not provide sufficient evidence to reject the test (p-value=0.6145). Hence the throughput performance generated by the GPU is similar to the one by CPU.

In this Chapter, the results of the simulations were presented relative to agent flow or throughput. For the model configurations chosen (environment size and shape, agent rules, etc.), the ACO and LEM based simulations performed similarly for lower number of agents, and the ACO based simulation performed much better than the LEM based simulation when the number of agents was fairly large. This in turn lends credibility to

the hypothesis that the additional information left by the predecessor agents for the following agents, replicating the visual indication in a real world scenario, produces a better result than the LEM based model. The viability of the model implemented on GPU was also supported by the comparison to the CPU based model, showing no statistically significant differences in throughput results.

CHAPTER 8

In this concluding chapter, a summary of the thesis contribution is made as well as suggestions for future work.

8.1 Conclusion

This thesis investigated the movement of pedestrians in an agent-based modeling approach. Two primary modeling methods were explored: LEM and ACO. In the LEM model, agents or pedestrians try to move toward their target taking the least-distance path towards the target, making as little deviation as possible from a near optimal path. In the modified ACO simulation, agents have some additional intelligence. In an ACO simulation agents can move by following their predecessor using a pheromone trail in conjunction with a least effort mode.

Through the simulation, it is demonstrated that using ACO based simulation produces better agent throughput for higher crowd densities and is considered to be a better pedestrian model result when compared to the LEM based model. In both cases, the LEM and ACO were run on a GPU to demonstrate that the GPU implementation is a natural computing environment for modeling movements in space that essentially involve local communication or interaction. Results produced by the modified ACO based simulation model surpassed the LEM based model by 39.6% in overall throughput aggregated over

all simulation scenarios. However, performance between the ACO and LEM based approaches varied widely based on population density, with the ACO outperforming the LEM relative to agent throughput at higher population densities.

In terms of overall throughput, the LEM and ACO based simulations produce virtually identical throughput results when the pedestrian density is low. The ACO provides for more optimal paths when the density is medium, and when highly congested neither the LEM nor ACO offer a means for pedestrian movement. This significant increment in throughput is obtained with a minimal increment of 11% in the execution time in the ACO based simulation compared to the LEM counterpart.

The simulation results also indicate that the ACO model simulated on both the CPU and GPU are similar to one another, adding credibility to the quality of the solution obtained from the GPU platform. This work demonstrated the techniques adopted to avoid warp divergences and achieve substantial speedup with a large number of pedestrians in the simulations. Index mapping techniques were adopted, such as loading of the edge elements from the neighbor tiles which could also be helpful in other applications such as image processing. From the above discussion it can be concluded that the ACO model is likely a more effective model for large crowd simulation purposes than the LEM model. This result bolsters the confidence in using an ACO model to simulate a real world scenario such as a mass gathering. The usefulness of these results adds to the tools and techniques that can be used to undertake more significant modeling efforts, such as those aimed at crowd management and safety. As anticipated, the parallel GPU implementation produces pedestrian throughput results that are similar to the single thread CPU based model in terms of model accuracy.

8.2 Future Work

Utilizing data parallelism and using a metaheuristic algorithm to emulate social dynamics is still in its preliminary stages and needs further investigation. While implementing the above algorithms, many simplifying assumptions were made. The velocity and size of the pedestrians were kept constant in all the simulations performed. In the next phase, one of the challenges will be to introduce more complexity into pedestrian characteristics while maintaining data parallelism. Another objective is to introduce a panic to emulate some sort of crisis situation. Furthermore, separating the scanning ranges and moving ranges of the pedestrians would be an interesting feature to introduce and simulate. Currently, the scanning range and the movement range of the pedestrians are confined only to the neighboring cells. Increasing the scanning range as well as the movement range and using different values for scanning and moving ranges to make decisions would be more practical and would add realism to the simulation.

On the hardware side, several enhancements can be considered. The GPU that is used has the FERMI architecture. Using the more recent KEPLER architecture [56] with advanced features would add to the performance. CUDA streams on FERMI GPU actually operate in a queue. However, on Kepler GPUs, they can be launched as actual separate streams in parallel. Using advanced GPUs will also provide more CUDA cores along with more memory for all stages. Notwithstanding future enhancements, the present work with its corresponding techniques can be considered a novel approach to using nature inspired metaheuristics for emulating a large number of pedestrians on GPUs utilizing data parallelism.

References

[1] D. Helbing, I. Farkas, P. Molnar, T. Vicsek, "Simulation of pedestrian crowds in normal and evacuation situations," in *Pedestrian and Evacuation Dynamics*, pp.21-58,Springer 2002.

[2] T.I. Lakoba, D.J. Kaup, N.M. Finkelstein, "Modifications of the helbing-molnar-farkas-vicsek social force model for pedestrian simulation", *Simulation*, Vol. 81, No. 5, pp. 339-352, 2005.

[3] D. Helbing, "A Mathematical Model for the Behavior of Pedestrians," *Behavioral Science*, Vol. 36, No. 4, pp.298-310, 1991.

[4] H. Klupfel, "The simulation of crowds at very large events," in *Traffic and Gran ular Flow '05*, Springer 2005.

[5] Y. Weifeng, T. K. Hai, "A novel algorithm of simulating multi-velocity evacua tion based on cellular automata modelling and tenability condition," *Physica A: Statistical Mechanics and its Applications,* Vol. 379, No. 1, pp. 250-262, 2007.

[6] C. Burstedde, A. Kirchner, K. Klauck, A. Schadschneder, J. Zittartz, "Cellular automata approach to pedestrian dynamics - applications", arXiv:cond-mat/0112119.

[7] K. Nagel, M. Schreckenberg, "A cellular automation model for freeway traffic," Journal De Physique, Vol. 2, No. 12, pp. 2221-2229, Dec. 1992.

[8] S. Wolfram, *A New Kind of Science*, Wolfram Media, Inc., May 14 2002.

[9] S. Hoogendoorn, P. H. L. Bovy, "Gas-Kinetic Modeling and Simulation of Pedes trian Flows," *Journal of Transportation Research Board*, Vol. 1710, No.1, pp. 28-36, January 2010.

[10] F. Cherif, R. Chighoub, "Crowd simulation influenced by agent's socio-psychological state," *Journal of Computing*, Vol. 2, No. 4, April 2010.

[11] F. Durupinar, J. M. Allbeck, N. Pelechano, N. I. Badler, "Creating crowd varia tion with the ocean personality model.," in Proc. Of *International Foundation for Autonomous Agents and Multiagent Systems* (IFAAMAS), pp. 1217–1220, 2008.

[12] S. R. Musse, D. Thalmann, "A model of human crowd behavior: Group interrela tionship and collision detection analysis," in *Workshop on Computer Animation and Simulation of Eurographics*, pp. 39–52, 1997.

[13] M. Sung, M. Gleicher, S. Chenney, "Scalable behaviors for crowd simulation," *Computer Graphics Forum*, Vol. 23, No. 3, pp. 519–528, 2004.

[14] N. Fridman, G. A. Kaminka, "Towards a cognitive model of crowd behavior based on social comparison theory," *Association for the Advancement of Artificial Intelligence (AAAI)*, pp. 731–737, 2007.

[15] S. Sarmady, F. Haron, A.Z.H. Talib, "Modelling Groups of Pedestrians in Least Effort Crowd Movements Using Cellular Automata," *IEEE Symp. on Modelling and Simulation* (AMS2009), pp 520–525, May 2009.

[16] M. Dorigo, T. Stützle, "Ant Colony Optimization," MIT Press, Cambridge, MA,2004.

[17] M. Dorigo, V. Maniezzo, A. Colorni, "The ant system: optimization by a colony of cooperating agents," *IEEE Trans. on Systems, Man, and Cybernetics, Part B*, Vol.26, pp. 29–41, 1996.

[18] C. Blum, "Ant Colony Optimization: introduction and recent trends," *Physics of Life Reviews 2*, Vol. 2 No. 4, pp 353-373, 2005.

[19] E. Lawler, J.K Lenstra, A.H.G R. Kan, D.B Shmoys., "The travelling salesman problem," John Wiley & Sons, New York, 1985.

[20] L.M. Gambardella, E.D. Taillard, G. Agazzi, "MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows," in *New Ideas in Optimization*, McGraw Hill, London, UK, pp. 63–76,1999.

[21] J. Zhang, H.S.H. Chung, A.W.L. Lo, T. Huang, "Extended ant colony optimiza tion for power electronic circuit design," *IEEE Trans. on Power Electronics*, Vol. 24, No.1, pp. 147-162, Jan 2009.

[22] A. Shmygelska, H.H Hoos, "An ant colony optimisation algorithm for the 2D and 3D hydrophobic polar protein folding problem," *BMC Bioinformatics*, Vol. 6, No: 30, 2005.

[23] D. Kirk, W.M. W. Hwu, Programming Massively Parallel *Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[24] J. Sanders, E. Kandrot, *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, July 2010.

[25] NVIDIA, "CUDA C Programming Guide V5.5," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2013.

[26] J. Milazzo, N. Rouphail, and D. P. Allen, "Effect of pedestrians on capacity of sig nalized intersections," *Transportation Research Record*, Vol. 1646, pp. 37–46, 1998.

[27] D. Helbing, "A fluid-dynamic model for the movement of pedestrians," *Com plex Systems*, Vol. 6, pp. 391–415, 1992.

[28] R. L. Hughes, "The flow of human crowds," *Annual Review of Fluid Mechanics*, Vol. 35, No. 1, pp. 169–182, 2003.

[29] M. Rose Challenger, *Understanding Crowd Behaviours*, Cabinet Office Emer gency Planning College, 2009.Vol. 35, No. 1, pp. 169–182, 2003.

[30] O. Beltaief, S. E. Hadouaj, K. Ghedira, "Multi-agent simulation model of pedestri ans crowd based on psychological theories," *4th International Conference on Logistics (LOGISTIQUA)*, pp.150–156, May 31-June 3, 2011.

[31] E. Bonabeau, "Agent-based modeling: methods and techniques for simulating hu man systems.," *Proc. of the National Academy of Sciences of the United States of America*, Vol. 99, No. 3, pp. 7280–7287, May 2002.

[32] V. Blue, J. Adler, "Cellular automata model of emergent collective bi-directional pedestrian dynamics," *In Proc. Artificial Life VII*, pp. 437—445, August 2000.

[33] A. Schrijver, "A Course in Combinatorial Optimization".

[34] L.J. Fogel, A.J. Owens, M.J. Walsh, *Artificial Intelligence Through Simulated Evolution*, John Wiley & Sons, 1966.

[35] V. Cerny, "A thermodynamical approach to the traveling salesman problem," *Journal of Optimization Theory and Applications*, Vol. 45, No. 1, pp. 41–51, 1985.

[36] F. Glover, "Tabu search—part I," *ORSA Journal on Computing*, Vol. 1, No. 3, pp. 190–206, 1989.

[37] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv.* , Vol. 35, No. 3, pp. 268-308, September 2003.

[38] P.P. Grass´e, *Les Insectes Dans Leur Univers*, Paris, France: Ed. du Palais de la d´ecouverte, 1946.

[39] M. Dorigo; M. Birattari; T. Stutzle, "Ant colony optimization," *IEEE ComputationalIntelligence Magazine*, vol.1, no.4, pp.28,39, Nov. 2006.

[40] T. St¨utzle, H.H. Hoos, “The *MAX–MIN* Ant System and local search for the traveling salesman problem,” in *IEEE International Conference on Evolutionary Computation (ICEC’97)*, pp. 309–314, 1997.

[41] M. Reimann, K. Doerner, R.F. Hartl, “D-ants: Savings based ants divide and Con quer the vehicle routing problem,” *Computers & Operations Research*, Vol. 31, No. 4, pp. 563–591, 2004.

[42] V. Maniezzo, “Exact and approximate nondeterministic tree-search procedures for The quadratic assignment problem,” *INFORMS Journal on Computing*, Vol. 11, No. 4, pp. 358–369, 1999.

[43] D. Merkle, M. Middendorf, H. Schmeck, “Ant colony optimization for resource Constrained project scheduling,” *IEEE Transactions on Evolutionary Computa tion*, Vol. 6, No. 4, pp. 333–346, 2002.

[44] D. Helbing, P. Molnar, I.J. Farkas, K. Bolay, “Self-organizing pedestrian move ment”, *Journal of Environment and Planning B: Planning and Design*, Vol. 28, No. 3, pp. 361-383, 2001.

[45] NVIDIA, “*NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*”.

[46] NVIDIA, “*CUDA C BEST PRACTICES GUIDE*”, July 2013.

[47] Y. Liu**,** D. L Maskell**,** B. Schmidt, “CUDASW++: optimizing Smith-Water man sequence database searches for CUDA-enabled graphics processing units”, *BMC Research Notes*, Vol. 2, No. 73, May 2009.

[48] NVIDIA, “*CUDA Occupancy Calculator V5.1*”.

[49] NVIDIA, “*CUDA COMPILER DRIVER NVCC V5.5*,” July 2013.

[50] J.M. Cecilia, J.M. Garcia, A. Nisbet, M. Amos, M. Ujaldon, “Enhancing data par allelism for ant colony optimization on GPUs,” *J. Parallel Distributed Compu ting*, Vol. 73, No. 1, pp. 42–51, 2013.

[51] TSPLIB Webpage, http://comopt.ifi.uniheidelberg.de/software/TSPLIB95/, Feb ruary 2011.

[52] S.M. Niaz Arifin, G.J. Davis, and Y. Zhou. "Verification & validation by dock ing: a case study of agent-based models of Anopheles gambiae." In *Proceedings of the 2010 Summer Computer Simulation Conference*, pp. 236-243. Society for Computer Simulation International, 2010.

[53] J. K. Lindsey, *Applying Generalized Linear Models*, New York: Springer, 1997.

[54] NVIDIA, *NVIDIA CUDA CURAND Library*, 2013.

[55] T. Scavo, "Scatter-to-gather transformation for scalability", https://hub.vscse.org/resources/223, August 2010.

[56] NVIDIA, "*NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 V1.0*".

Glossary

1. CPU : Central Processing Unit
2. GPU : Graphics Processing Unit
3. ACO : Ant Colony Optimization
4. LEM : Least Effort Model
5. ABM : Agent Based Model
6. CUDA : Compute Unified Device Architecture

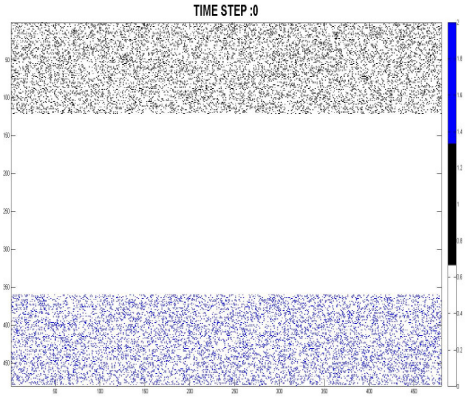
Appendix

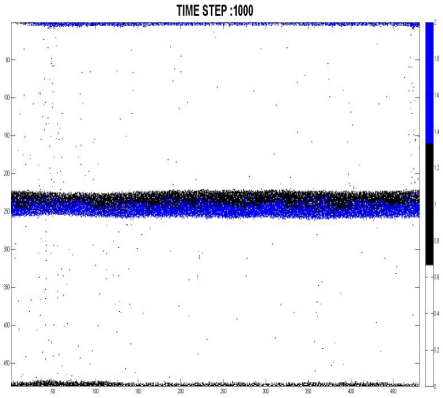
A1. Simulation pictures

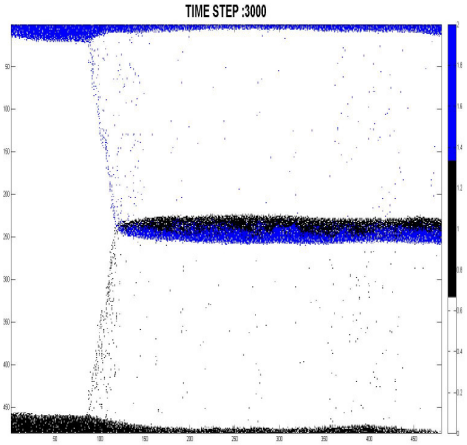
The pictures of the throughput of the pedestrian are presented in this section pictures of both LEM and ACO based simulation are depicted. All the pictures shown are for simulations carried on GPU. Here the simulation pictures that are pertinent with respect to the throughput are shown. In section A1.1, with 17,920 agents, the throughput of both the LEM and ACO based simulation are similar. However as the number of agents increases, the throughput of the ACO based model is better than the LEM based model which can be seen in section A1.2, A1.3 and A1.4. When the number of pedestrian reaches a high value of 40,960 the throughput deteriorates for both ACO and LEM as shown in section A1.5. However, still from the simulation pictures it can be seen that the throughput of ACO is better than LEM driven simulation.

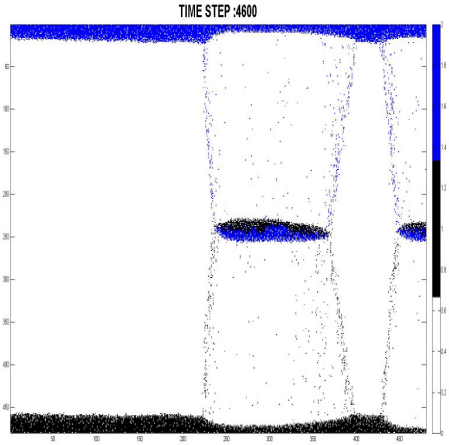
A1.1 Simulation Scenario Number 1 Using LEM and ACO With 17,920 Pedestrians

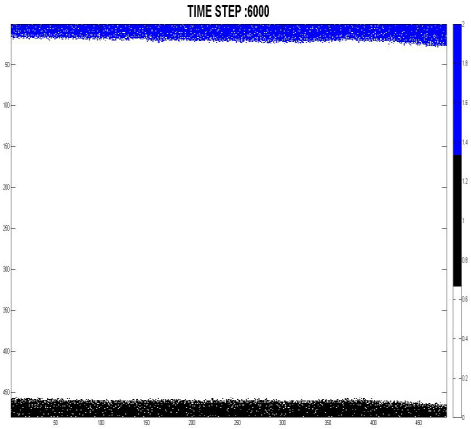
LEM Driven Simulation Pictures on GPU With 17920 Pedestrians

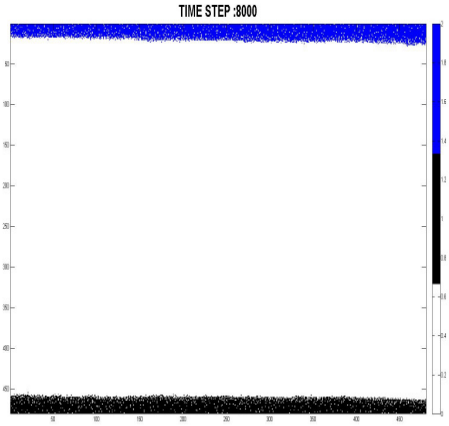




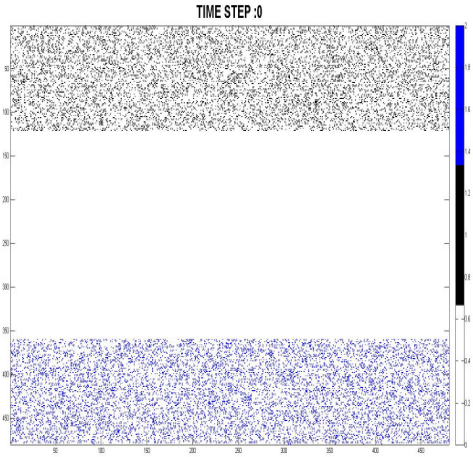


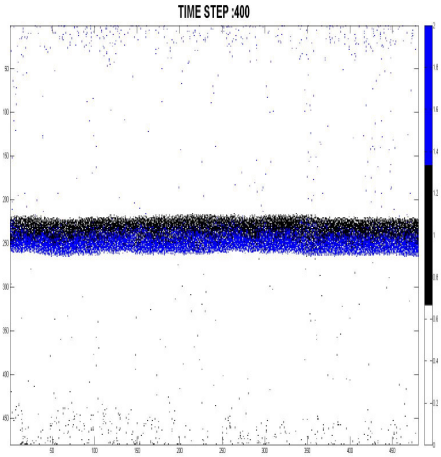


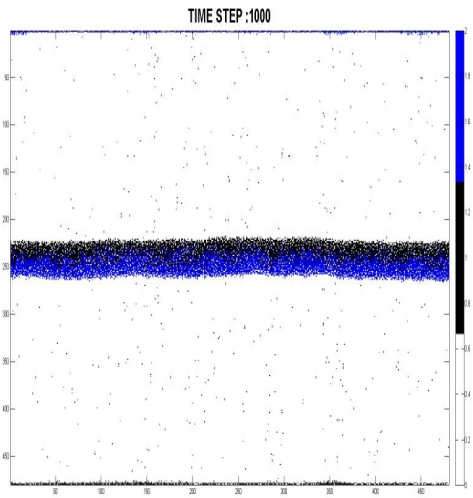


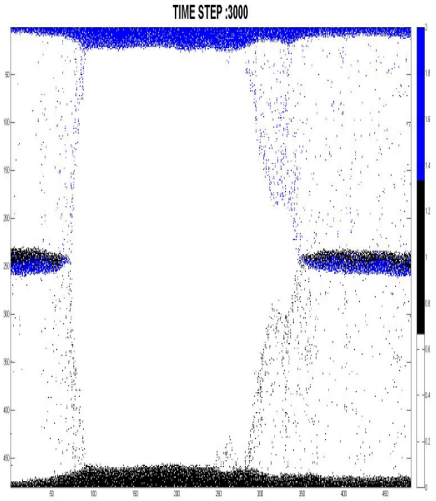


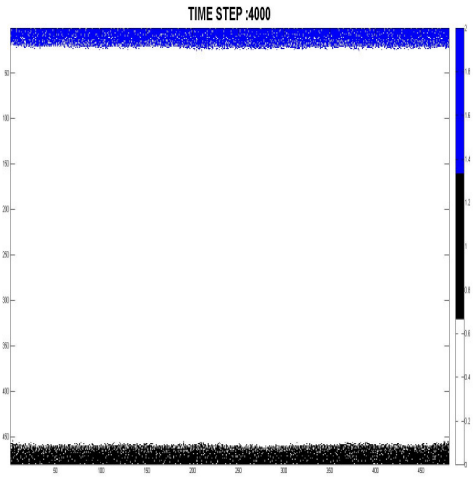
ACO Driven Simulation Pictures on GPU With 17920 Pedestrians

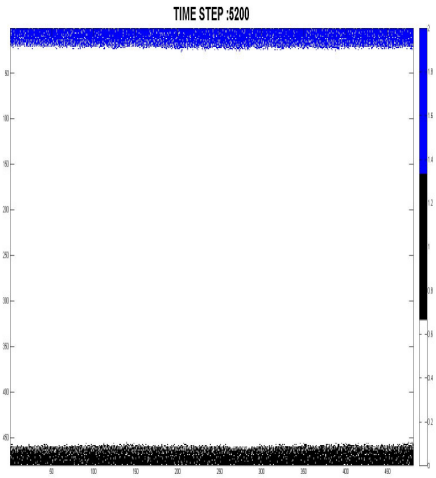






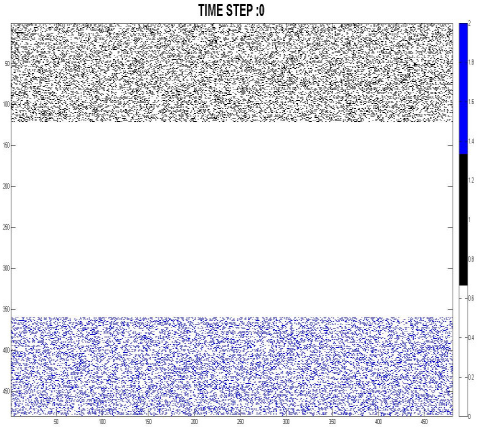


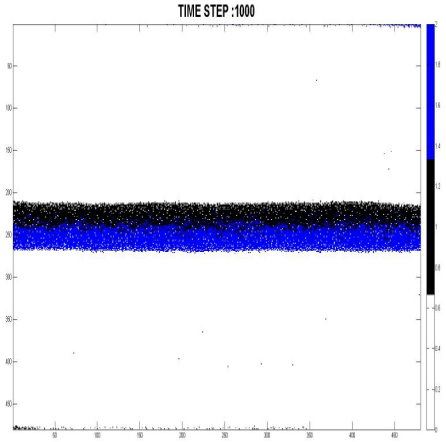


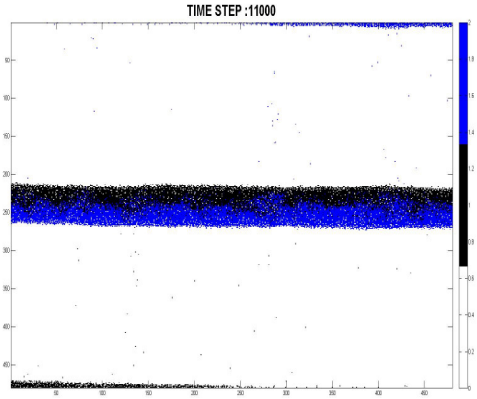


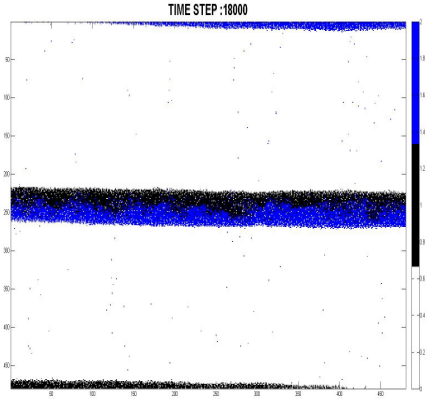
A1.2 Simulation Scenario Number 2 Using LEM and ACO With 25,600 Pedestrians

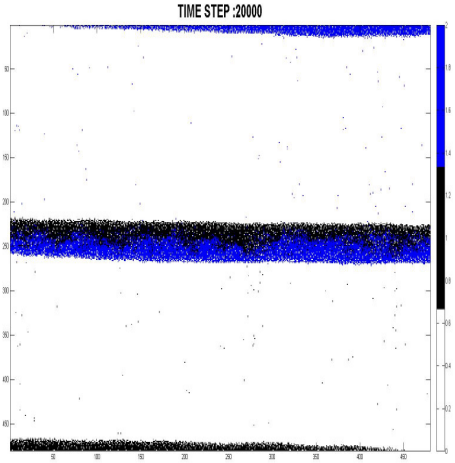
LEM Driven Simulation Pictures on GPU with 25,600 Pedestrians

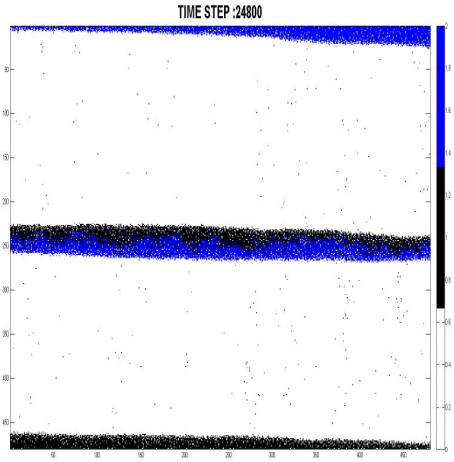




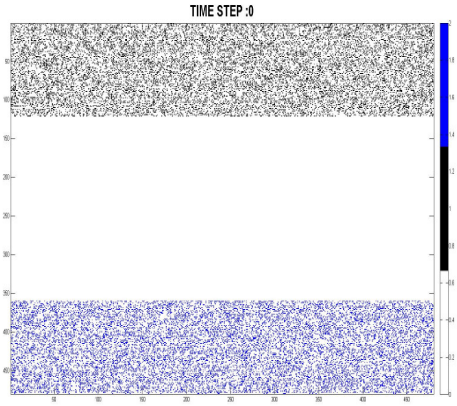


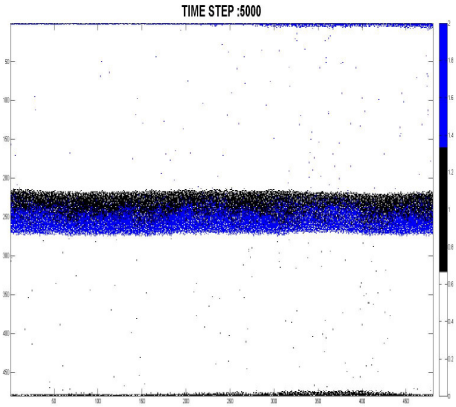


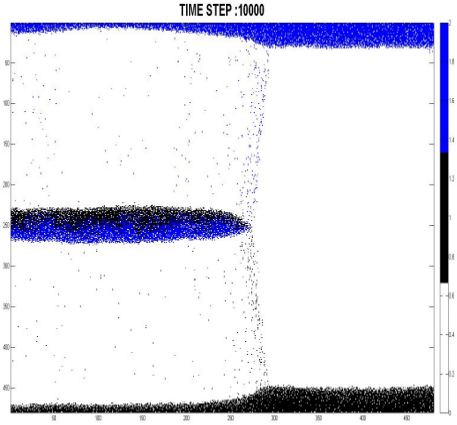


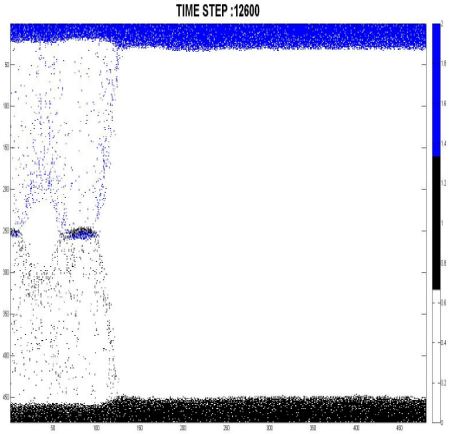


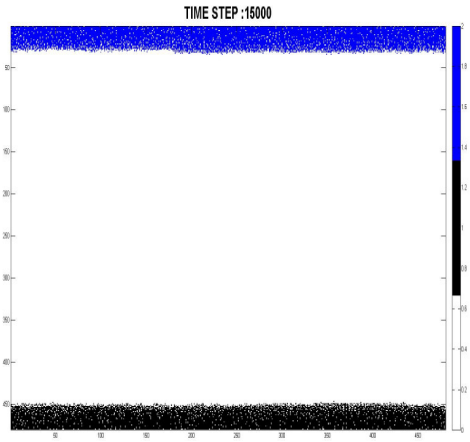
ACO Driven Simulation Pictures on GPU with 25,600 Pedestrians

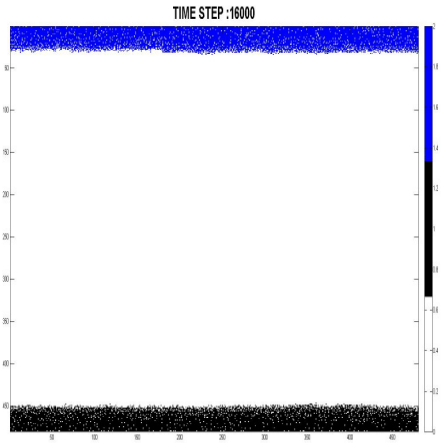






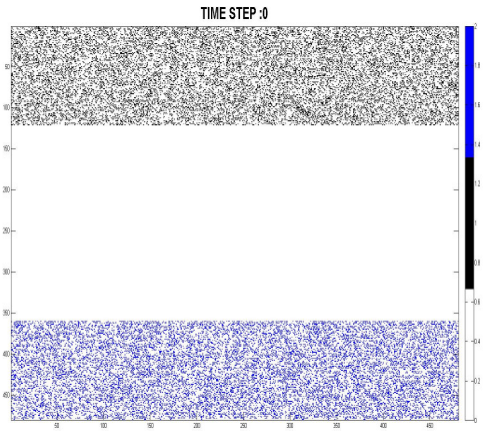


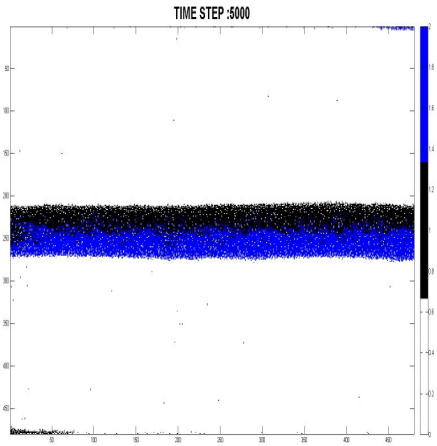


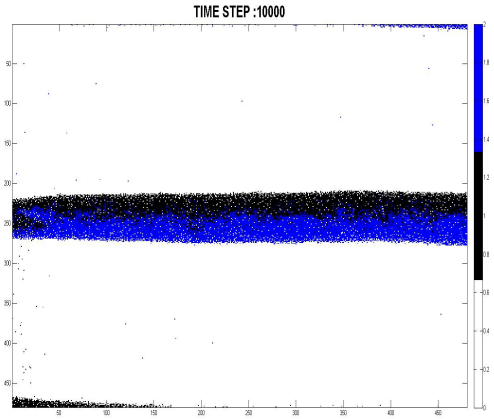


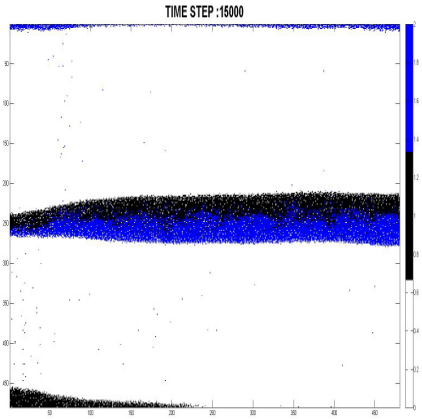
A1.3 Simulation Scenario number 3 Using LEM and ACO With 28,160 Pedestrians

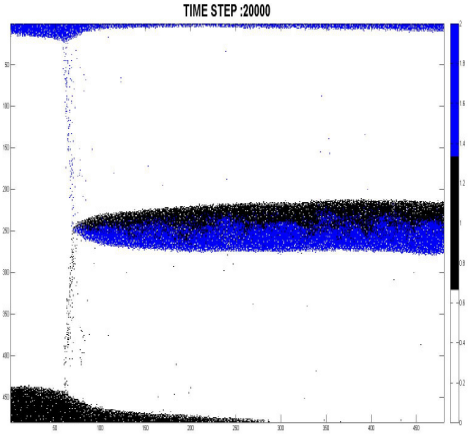
LEM Driven Simulation Pictures on GPU with 28,160 Pedestrians

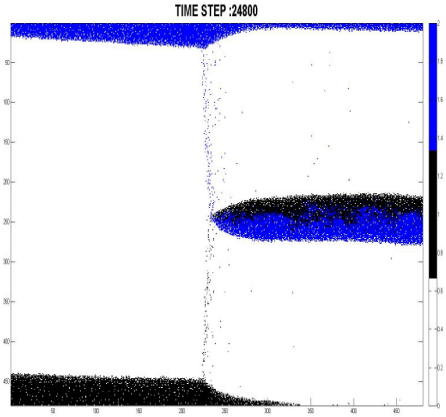




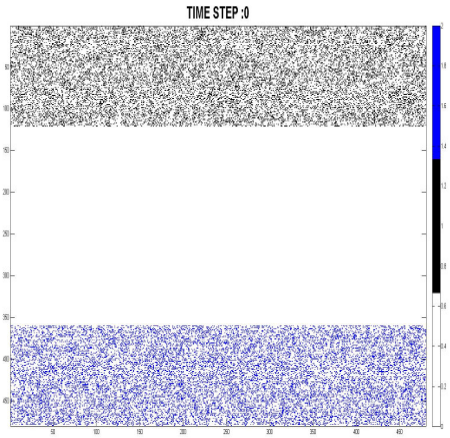


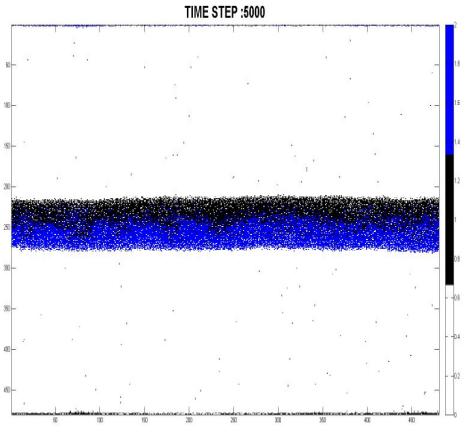


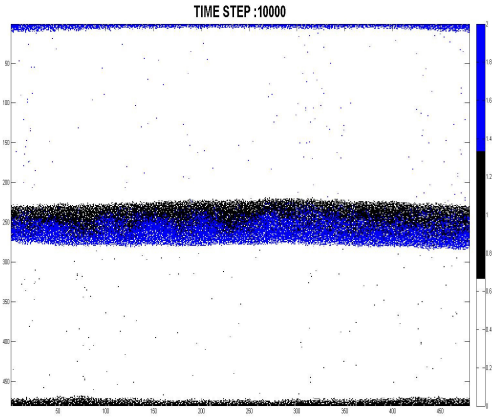


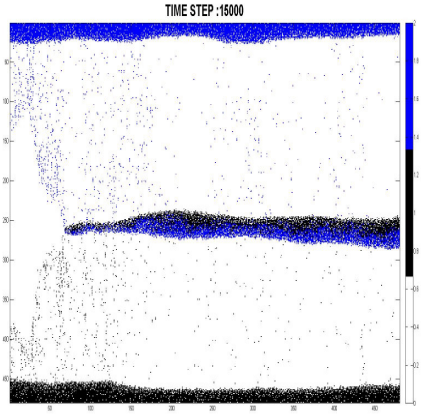


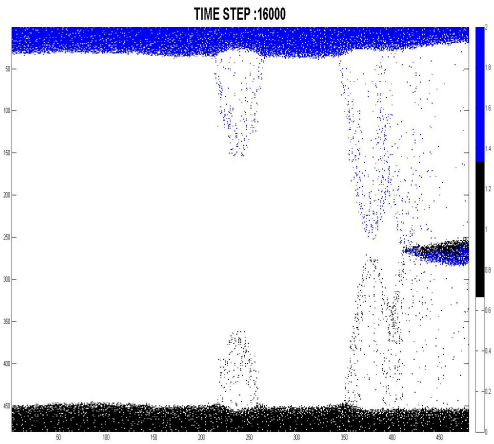
ACO Driven Simulation Pictures on GPU with 28,160 Pedestrians

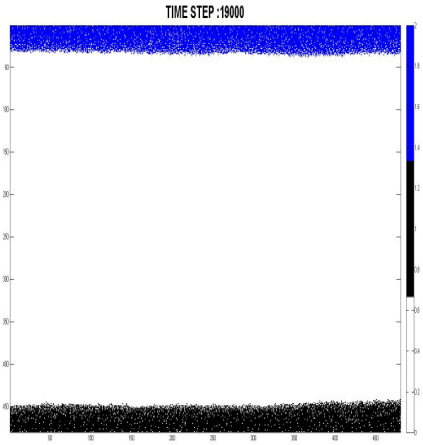






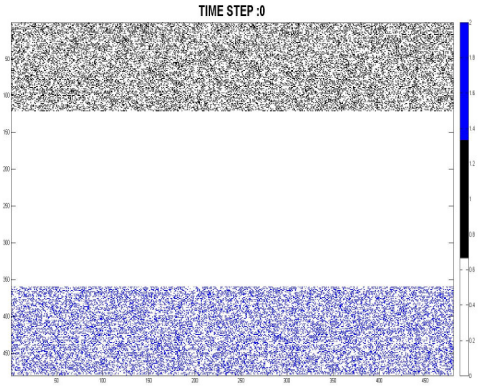


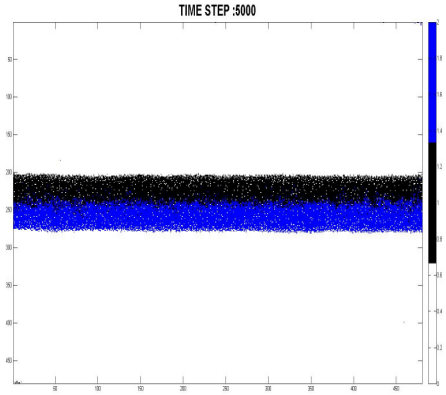


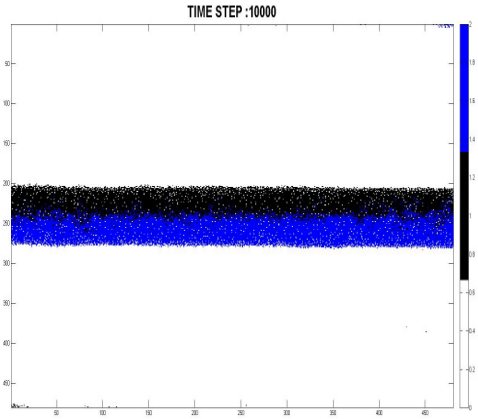


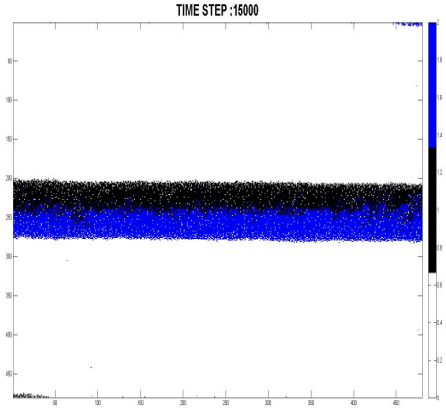
A1.4 Simulation Scenario number 4 Using LEM and ACO With 33,280 Pedestrians

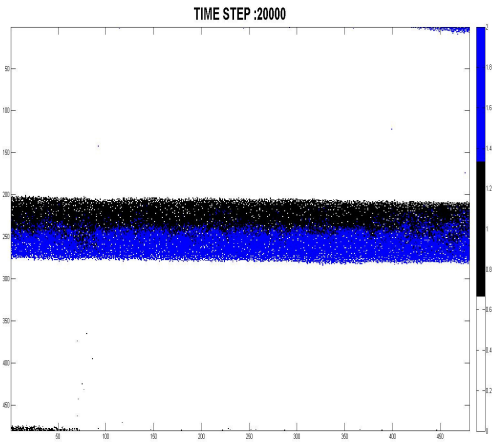
LEM Driven Simulation Pictures on GPU with 33,280 Pedestrians

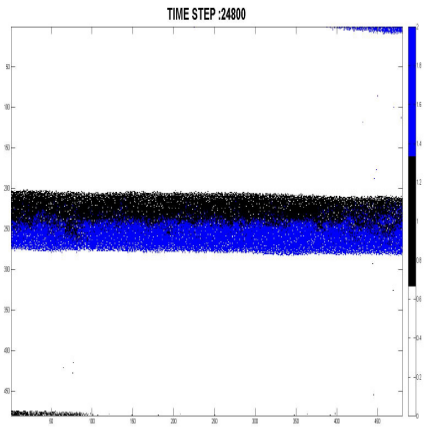




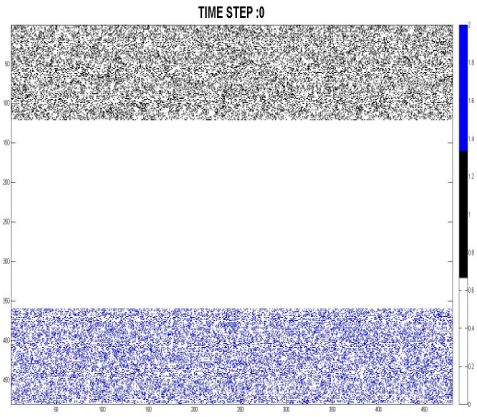


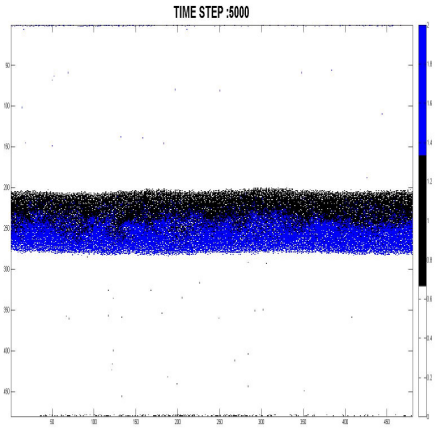


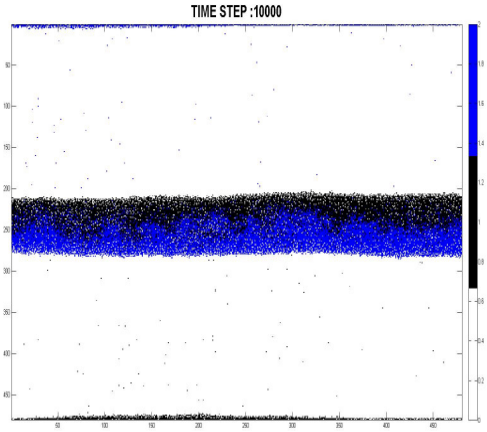


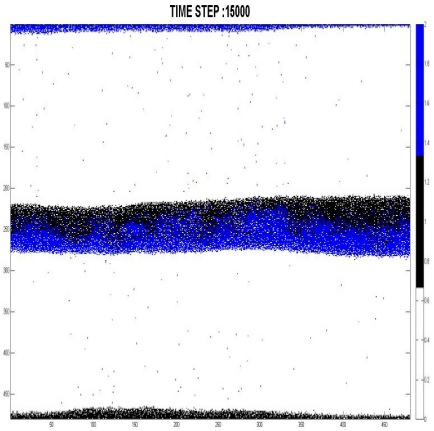


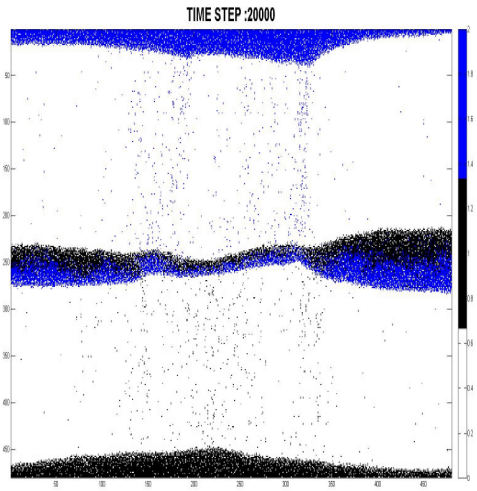
ACO Driven Simulation Pictures on GPU with 33,280 Pedestrians

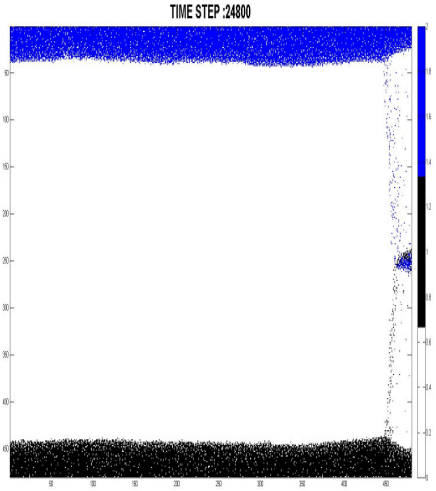






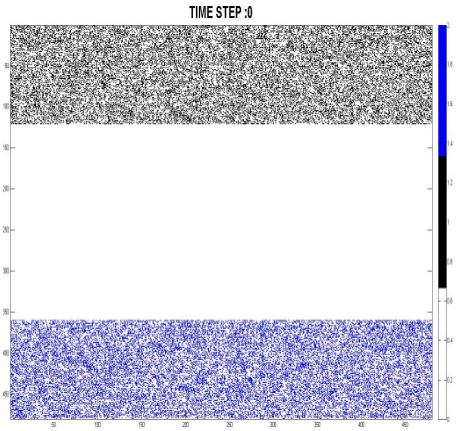


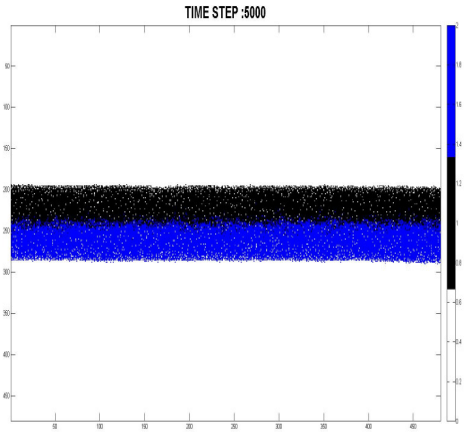


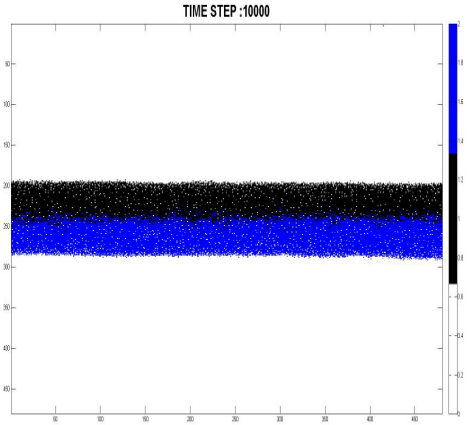


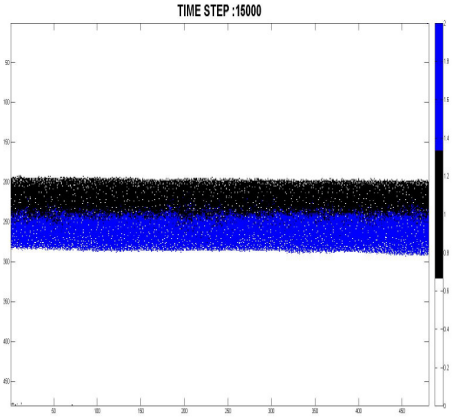
A1.5 Simulation Scenario number 5 Using LEM and ACO With 40,960 Pedestrians

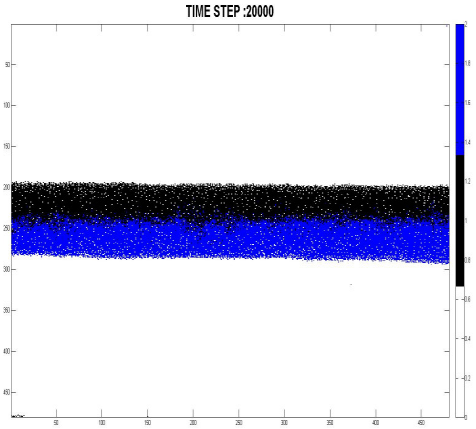
LEM Driven Simulation Pictures on GPU with 40,960 Pedestrians

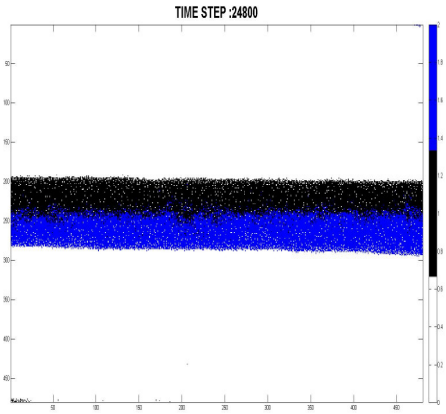












ACO Driven Simulation Pictures on GPU with 40,960 Pedestrians