Pertanika J. Sci. & Technol. 26 (3): 1019 - 1038 (2018)

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

GPU-based Optimization of Pilgrim Simulation for Hajj and Umrah Rituals

Abdur Rahman Muhammad Abdul Majid[1], Nor Asilah Wati Abdul Hamid[1]*, Amir Rizaan Rahiman[1] and Basim Zafar[2]

[1]Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 UPM, Serdang, Selangor, Malaysia

[2]Custodian of the Two Holy Mosques Institute for Hajj Research, Umm Al-Qura University, Makkah, Saudi Arabia

ABSTRACT

Tawaf ritual performed during Hajj and Umrah is one of the most unique, large-scale multi-cultural events in this modern day and age. Pilgrims from all over the world circumambulate around a stone cube structure called Ka'aba. Disasters at these types of events are inevitable due to erratic behaviours of pilgrims. This has prompted researchers to present several solutions to avoid such incidents. Agent-based simulations of a large number of pilgrims performing different the ritual can provide the solution to obviate such disasters that are either caused by mismanagement or because of irregular event plans. However, the problem arises due to limited parallelisation capabilities in existing models for concurrent execution of the agent-based simulation. This limitation decreases the efficiency by producing insufficient frames for simulating a large number of autonomous agents during Tawaf ritual. Therefore, it has become very necessary to provide a parallel simulation model that will improve the performance of pilgrims performing the crucial ritual of Tawaf in large numbers. To fill in this gap between large-scale agent-based simulation and navigational behaviours for pilgrim movement, an optimised parallel simulation software of agent-based crowd movement during the ritual of Tawaf is proposed here. The software comprises parallel behaviours for autonomous agents that utilise the inherent parallelism of Graphics Processing Units (GPU). In order to implement the simulation software, an optimized parallel model is proposed. This model is based on the agent-based architecture which comprises agents having a reactive design that responds to a fixed set of stimuli. An advantage of using agents is to provide artificial anomaly to generate heterogeneous movement of the crowd as opposed to a singular movement which is unrealistic. The purpose is to decrease the execution time of complex behaviour computation for each agent while simulating a large crowd of pilgrims at increased frames per second (fps). The implementation utilises CUDA

ARTICLE INFO

Article history:
Received: 30 May 2017
Accepted: 17 January 2018

E-mail addresses:
arahman.ned@gmail.com (Abdur Rahman Muhammad Abdul Majid)
asila@upm.edu.my (Nor Asilah Wati Abdul Hamid
amir_r@upm.edu.my (Amir Rizaan Rahiman)
bjzafar@uqu.edu.sa (Basim Zafar)
*Corresponding Author

ISSN: 0128-7680 © 2018 Universiti Putra Malaysia Press.

(Compute Unified Device Architecture) platform for general purpose computing over GPU. It exploits the underlying data parallel capability of an existing library for steering behaviours, called OpenSteer. It has simpler behaviours that when combined together, produces more complex realistic behaviours. The data-independent nature of these agent-based behaviours makes it a very suitable candidate to be parallelised. After an in-depth review of previous studies on the simulation of Tawaf ritual, two key behaviours associated with pilgrim movement are considered for the new model. The parallel simulation is executed on three different high-performance configurations to determine the variation in different performance metrics. The parallel implementation achieved a considerable speedup in comparison to its sequential counterpart running on a single-threaded CPU. With the use of parallel behaviours, 100,000 pilgrims at 10 fps were simulated.

Keywords: Agent-based system, crowd simulation, GPU, Hajj and Umrah, parallelisation

INTRODUCTION

Many scientific applications require high-performance computing systems to perform their computational tasks expeditiously. This can be done using Graphics Processing Unit (GPU), as it can solve compute-intensive tasks on thousands of highly parallel, multi-threaded processing cores. The GPU utilises different levels of memory and high data throughput to accelerate the computational process. The GPUs has given rise to a programmable ecosystem that started leveraging on its highly parallel architecture. NVIDIA® standardised the customisation with the release of a toolkit called *Compute Unified Device Architecture* (CUDA). This API is an extension of C programming language. This has provided researchers with the opportunity to port several general-purpose applications onto the GPU. This notion is referred to as *General-purpose Computing on Graphics Processing Unit* (GPGPU).

Among the range of applications being ported to GPU, agent-based crowd simulation remains the most computer-intensive application. This is due to the additional computational power required with the increase in the number of agents present in a virtual environment. Researchers have proposed several models to simulate crowd movement including social force models (Helbing & Farkas, 2002), cellular automata models (Klüpfel, 2007), gas kinetic models (Hoogendoorn & Bovy, 2001) and agent-based models (Cherif & Chighoub, 2010). Among these, agent-based modelling (ABM) (Jennings, 2000), is the more frequently used methodology to implement autonomous agents. In the case of religious rituals such as Hajj and Umrah, simulating such a large crowd adds to computational demand of the system. Since each agent in the simulation possesses complex behaviours, this provides a real-life crowd which depicts the unique crowd phenomenon in these rituals.

Hajj and Umrah are performed by thousands of pilgrims all over the world. In 2015, a stampede killed more than 2000 pilgrims (Salamati & Rahimi-Movaghar, 2016). Such disasters can be prevented if there was a simulation earlier. Earlier studies were not able to simulate a large crowd to possible outcomes (Kim et al., 2015; A. N. Shuaibu, Faye, Malik, & Talal, 2014; Sakellariou et al., 2014; Sarmady, Haron, & Talib, 2011).

Therefore, there is a dire need for an agent-based simulation that can replicate dense crowd scenarios and depict the natural movement of pilgrims. This research reports the behaviours on the GPU for concurrent execution. This was implemented by Reynolds (2006) which is an open source steering library called *OpenSteer*. The current work extends previous research Rahman, Hamid, Rahiman, & Zafar, (2015).

LITERATURE REVIEW

In the field of crowd simulation and modelling, some researchers have focused on Hajj and Umrah. AlGadhi et al. was among the first to predict the throughput of pilgrims from the ``Jamarat" site (AlGadhi & Mahmassani, 1990). His work provided the basis of pilgrim movement and their flow for future research. Abdelghany et al. proposed a micro-simulation based on the model provided by AlGhadi (Abdelghany, Abdelghany, Mahmassani, & Al-gadhi, 2006). They assessed the structure of Masjid Al-Haram by using three different levels of congestions. Zainuddin et al. simulated up to 1,000 pilgrims, implemented using proprietary software called ``SimWalk" (Zainuddin, Thinakaran, & Abu-Sulyman, 2009). Similarly, Mulyana et al. developed a 2-dimensional software that was able to represent 500 agents for Tawaf and Sa'yee rituals performed during Hajj (Mulyana & Gunawan, 2010). Rahim et al. (2011) developed one of the first t 3-dimensional simulations for Tawaf with visualisation capability of 500 agents. Using *Cellular Automata* model, Sarmady et al. (2011) were able to mimic the circular movements of 15,000 pilgrims performing Tawaf. *Finite State Machine* (FSM) was used to simulate 35,000 agents in 2D (Curtis, Guy, Zafar, & Manocha, 2013). In a more recent research, the spiral model was used by Shuaibu et al. (2013) to simulate 1000 agents . Using X-machine model with NetLogo, Sakellariou et al. (2014) were able to represent results for 1500 primitive agents performing Sa'yee in Masjid Al-Haram. These studies faced certain limitations regarding numbers of virtual pilgrims because of limited computational power. In other words, the CPU was incapable of providing the required computational power to display the complex behaviour of pilgrims at interactive frame rates.

Therefore, there is a research gap between crowd simulation in Hajj and Umrah rituals, and the performance optimisation techniques used. Although many studies have examined Tawaf simulation, none of them addressed the performance of these simulations. The need to compute complex behaviours of a large virtual crowd is in line with the increasing number of pilgrims attending the religious event of Hajj each year (see Figure 1). Hence, to provide the required performance and computation for simulating large crowd in Hajj, the study proposes a Parallel *Agent-based crowd simulation*. It also shows some of the common behaviours observed among pilgrims during Hajj and Umrah rituals. This research will take advantage of the underlying capabilities of a GPU.

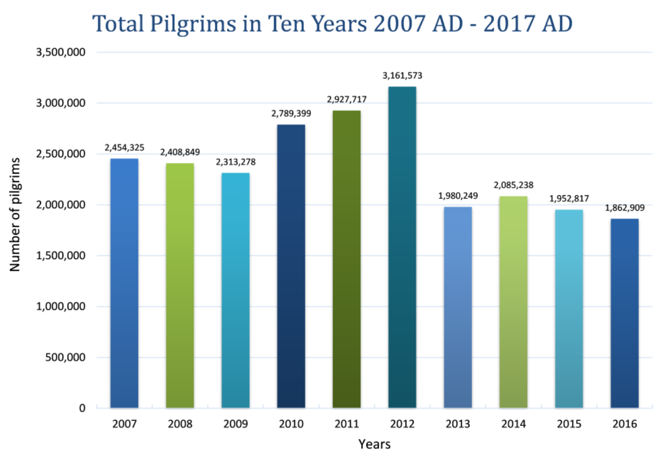

Figure 1. Number of pilgrims from 2007 – 2016 (Gen. Auth. for Stat. Saudi Arabia, 2016)

AGENT MODEL

In this research, it is necessary to design an agent model to comprehend the working of the proposed parallel agent-based simulation software. Each agent in the simulation acts as a building block for an agent-based system. Hence the understanding of their model is important before we proceed to the analysis of OpenSteer library.

The agent model is based on the reactive architecture in which is the response of the agent is based on fixed stimuli. The stimuli in this research include neighbouring agents, obstacles and the desired goal. The agent model for the proposed parallel implementation is shown in Figure 2.

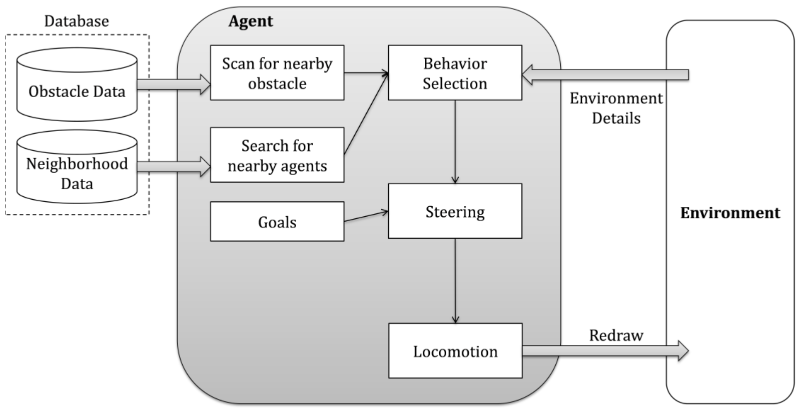

Figure 2. Agent model

In designing the agent model, the three main components of the agent includes behaviour selection, steering, and locomotion. The agent is intended to receive information about the environment. The behaviour selection component receives this information and decides on the behaviour available to use. This is also influenced by the obstacles and other agents in the vicinity. The selection of obstacle and agents to avoid will be performed on distance threshold value. This value provides the agent with ample time for determining an appropriate behaviour.

When the initial component selects the necessary behaviors, the steering part determines the underlying details. One of the most important detail that is selected is the steering force with the agent to proceed towards its goals. The information on the final goals is also provided which influences the force and truncate it accordingly.

The final module is referred to as locomotion. In this constituent, the steering force for the preceding component is utilised to calculate further properties of the agent. These properties include the velocity, orientation, and location. These properties will be used to redraw the agent in the next frame for representing the virtual environment. The following discussion provides a detailed analysis of OpenSteer library.

METHODOLOGY

The architecture of OpenSteer provides developers the ability to create complex behaviours by combining simpler behaviours within a scenario. These scenarios are referred as *plugins*. Before a plugin can starts simulating, each agent in the environment is first initialised. In this step, the objects for each agent is created, and other components are initialised (open ()). Then the created plugin is called. This call is placed in a loop, where each loop represents one simulation cycle. The first part of this step is known as the update () function. In this stage, calculations for each agent takes place and the second part is the redraw () function. The second function redraws each agent in at its new position. Ideally, this cycle should run 8 to 15 times in one second for the agents to move smoothly across the environment. At the end of the simulation, the next step (close ()) is activated which cleans up all the memory as shown in Figure 2.

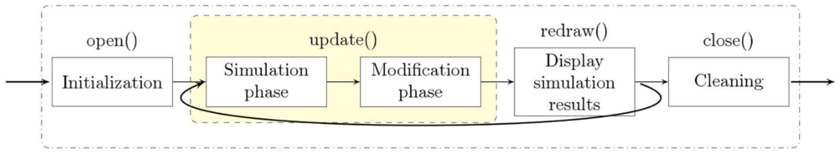

Figure 3. Simulation cycle (Rahman et al., 2015)

In OpenSteer library, the update function comprises two main parts, the *Simulation phase*, and the *Modification phase*. In the Simulation phase of the function, a *Steering Force* is calculated for each agent. If the agent abruptly moves towards the target, this will not produce realistic movement. To produce natural movement patterns, the steering force directs the agent towards its goal. In the Modification phase, the steering force from the previous phase is used to calculate the remaining properties (acceleration, velocity and future position) of the agents. The sequential implementation of library executes the update function for each agent one by one. So, if the simulation contains a large number of agents, then the function will require a longer time to complete. After calculating properties for each agent, the results are then transferred to the redraw function. This creates a bottleneck, since the execution time for each simulation cycle increases.

The calculations in the Simulation phase do not depend on the Modification phase, so this type of data-independent tasks favours parallelisation onto GPU. The navigational behaviours of each agent determine the steering force in the first phase. For all the agents present in the environment the parallel implementation will launch n threads for n number of agents. Each thread will initially calculate the steering force and will synchronise before the launch of the section phase. This representation can be seen in Figure 3.

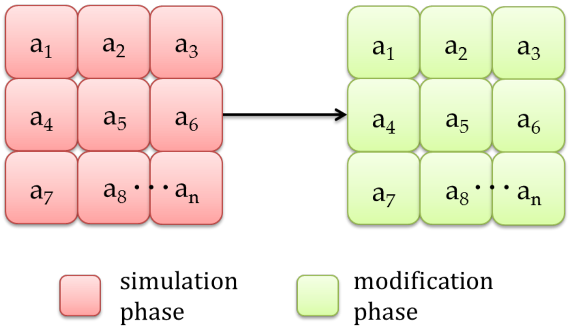

Figure 4. Traversing through simulation and modification phase in parallel

Two key navigational behaviours which are prominently seen in pilgrims performing rituals of Hajj and Umrah are selected. These behaviours include path following behavior and obstacle avoidance behavior. They allow agents to move freely in the virtual environment. The behaviours are very refined and are used to produce much of the natural motion of agents. The path following behaviour is essential for agents in the scenario of Hajj and Umrah. The complete path is considered as the goal of the character. Routes such as roads, streets, and footpaths are fixed. The character’s movement on the track is not fixed; rather a character moves freely along the path like a human. This movement is done near to the pathway keeping it as a goal. By utilising the obstacle avoidance behaviour, the agents can avoid accidents initially with other agents and secondly, with obstacles within the environment. The next section discusses parallel algorithms for the selected behaviours.

PARALLEL NAVIGATIONAL BEHAVIOR ALGORITHMS

In the sequential implementation of OpenSteer library, each agent has a set of preliminary behaviours that is selected based on the required action and situation. The behaviour calculates the steering force that drives an agent. Each of these behaviours independently calculates and updates the value of steering force for each agent. In this section, the parallel algorithms of the selected navigational behaviours are discussed.

Since the computation for each agent is independent, therefore, the behaviours can be easily converted into GPU-executable kernels. In order to convert this behavior into a kernel, memory management routines must be performed before invoking them. The specific parameters related to the vehicle are initially copied to the global memory. A predefined variable such as *threadIdx, blockIdx, blockDim*, and *gridDim* are used to access the memory space from within the kernels. When the required data is retrieved, only then the process for computing steering force begins. Path following is one of the selected behavior that an agent poses to follow a predefined path. This type of behaviour is commonly observed in pilgrims performing different rituals during Hajj and Umrah (Zafar, 2011). Algorithm 1 provides the implementation for this behaviour.

Algorithm 3 Parallel Path Following Behavior

```
$Input : vehicleData, steeringVectors, direction, predictionTime$
$Output : steeringVectors$
$id \leftarrow (blockId \times blockSize + threadId)$
$Velocity(threadId) \leftarrow ((float*)(*vehicleData).forward)[Offset + threadId]$
$threadSyncronize()$
$Velocity(threadId) \leftarrow Velocity(threadId) \times Speed(threadId)$
$Position(threadId) \leftarrow ((float*)(*vehicleData).position)[Offset + threadId]$
$threadSyncronize()$
$pathDistanceOffset \leftarrow direction[id] * predictionTime * Speed(threadId)$
$futurePosition \leftarrow PredictFuturePosition$
$nowPathDistance \leftarrow mapPointToPathDistance(points, totalPoints,$
    $Position(threadId))$
$futurePathDistance \leftarrow mapPointToPathDistance(points, totalPoints,$
    $futurePosition)$
if $pathDistanceOffset > 0$ then
    if $nowPathDistance < futurePathDistance$ then
        $rightway \leftarrow false$
    else if $nowPathDistance > futurePathDistance$ then
        $rightway \leftarrow true$
    end if
end if
$onPath \leftarrow mapPointToPath(points, totalPoints, radius, futurePosition,$
        $\&tangent, \&outside)$
if $(Position(threadId) - points[0]) < .radius$ then
    $direction[id] \leftarrow 1$
end if
if $(Position(threadId) - points[totalPoints - 1]) < radius$ then
    $direction[id] \leftarrow 1$
end if
if $(outside < 0)$ and $rightway$ then
    $target \leftarrow 0$
    $ignore \leftarrow 1$
else
    $targetPathDistance \leftarrow nowPathDistance + pathDistanceOffset$
    $target \leftarrow mapPathDistanceToPoint(points, totalPoints, isCyclic,$
            $targetPathDistance)$
    $ignore \leftarrow 0$
end if
$steerForSeekKernel(Position(threadId), Veocity(threadId), target,$
        $steeringVectors, ignore)$
```

Figure 5. Parallel Path following Algorithm

In the previous algorithm, it was observed that the steering force was calculated using the function for seeking behaviour. The function was transformed into a GPU-executable kernel, calling a function residing on the host is not possible. So, the seek behaviour function was changed into a callable device kernel using a unique function type qualifier provided by CUDA. The new function can be executed on a device, and it is only callable by the device. It stores the newly computed steering behavior back to the global memory of the GPU.

Obstacle and Collision avoidance behaviour is utilised to avoid oncoming obstacles and other agents. It is a natural behaviour for virtual agents when navigating through the environment. Algorithm 2 provided the working of the obstacle avoidance behavior.

Algorithm 4 Parallel Obstacle Avoidance Behavior

```
$Input : vehicleData, steeringVectors$
$Output : steeringVectors$
$id \leftarrow (blockId \times blockSize + threadId)$
$minDistanceToCollision \leftarrow (*vehicleData).speed[id] \times 5.f$
for $i \leftarrow 0, numOfObstacles$ do
    $findNextIntersectionWithSphere()$
    if $intersectionDistance < nearestIntersectionDistance$ then
        $nearestIntersectionDistance \leftarrow intersectionDistance$
        $nearestIntersectionID \leftarrow i$
    end if
end for
if $(intersectionFound = 1)$ and $(nearestIntersectionDistance < minDistanceToCollision)$ then
    $offset = position[id] - obstacles[nearestIntersectionID].center)$
    $avoidance(threadIdx) = PerpendicularComponent(offset, (*vehicleData).forward[id])$
    $avoidance(threadIdx) = avoidance(threadId) \times (*vehicleConst).maxForce[id]$
    $avoidance(threadIdx) = avoidance(threadIdx) + ((*vehicleData).forward[id] \times (*vehicleConst).maxForce[id])$
end if
$threadSynchronize()$
if $(steeringVectors[id].x \neq 0.f$ or $steeringVectors[id].y \neq 0.f$ or $steeringVectors[id].z \neq 0.f)$ then
    $avoidance(threadIdx) = steeringVectors[id]$
else
    $avoidance(threadIdx) = avoidance(threadIdx) + steeringVectors[id]$
end if
$threadSynchronize()$
$steeringVectors[id] \leftarrow avoidance(threadIdx)$
```

Figure 6. Parallel Collision avoidance Algorithm

The behaviour starts by measuring the minimum distance to the collision. In the sequential implementation, the minimum time to the collision is provided as a function parameter. However, in the parallel implementation, that value is fixed. This value helps make the process simpler rather having a different value each time. After the calculation for the avoidance is completed, the thread synchronisation is called to ensure that all the thread has computed the avoidance vector for all agents. When all threads have finished their calculation, the measured vector is copied back to the steering force, again using the id to place the data back uniquely into the allocated memory space. This data is then utilised by the redraw function to represent the agents in the virtual world.

Some necessary details that must be dealt with to ensure proper parallelisation. The first one was the change in phase traversal that is discussed earlier. The parallelised behaviours were blended together, each affecting the resultant steering force. This provides a more realistic movement for each agent. Each of these behaviours were invoked as CUDA kernels.

The second important thing to be considered is the change in data type that stores data for each agent. In the sequential implementation, the *Vec3* datatype was commonly used to store the three-dimensional coordinates of the agent. Since CUDA does not support this data type, the new datatype of *float3* is used. The *float3* data type is compatible with the CUDA runtime. It is of type struct and has three members *x, y*, and *z*. A built-in function make*_float3* converts the three components of the *Vec3* datatype to *float3 member variables*.

Another hurdle in the parallel conversion of the library is to carefully orchestrate the memory allocation and transfer sequences to execute the parallel algorithms properly. Attributes for all the agents needed to be copied to the GPU memory to execute the behaviour kernels developed in CUDA. Proper thread synchronisation and memory management techniques are used to ensure that the data is used for each agent.

With this, a parallel behavioral library is provided to developers, in which simpler behaviours can be combined to create complex behaviours.

KERNEL TYPES AND MANAGEMENT

During the transition from sequential to parallel implementation, two main kernels are introduced, namely, *Steering Kernel* and *Modification Kernel*. Steering kernel calculates the steering force for all agents, whereas the modification kernel applies the calculated steering force to the properties of every agent. In the parallel implementation, these kernels are used to determine the steering force of each pilgrim. In the update phase, the first *steering kernel* is responsible for describing how the pilgrim will navigate by providing an appropriate steering force, whereas, the second *modification kernel* uses the steering force value and measures the rest of the properties for the pilgrim. These include the *velocity, direction, speed* and other properties. For this purpose, an array of data type *float3* for the variable *steering Vectors* is allocated in the global memory of the GPU that stores the steering force for the total number pilgrims in each scenario. By transferring this data onto the GPU, each thread executing the kernel can read and update the data of pilgrims. So, each element of the array represents the steering force of one pilgrim. Figure 6 provides a visual representation of each steering kernel modifying the steering vector value. When the entire kernel has executed the final value, it is then acquired by the modification kernel to measure rest of the vehicle properties.

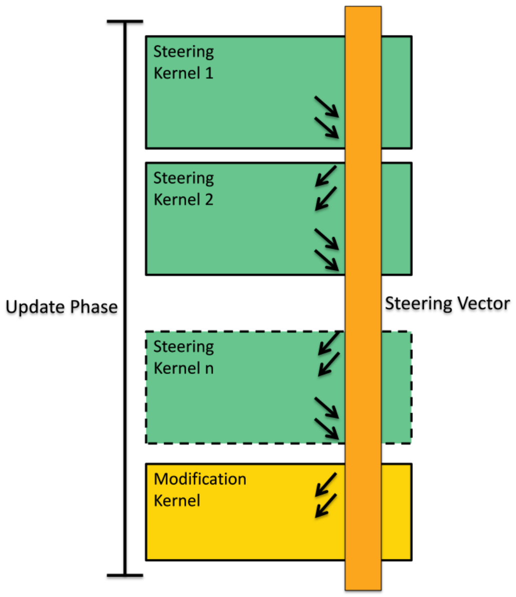

Figure 7. Steering vector accessed by different kernels

The modified data of the array containing the steering force needs to be cleared after every update cycle so the new steering forces can be stored in the next cycle. The navigational behaviours can be combined with each other to obtain more complex behaviours that can reflect more natural movement. There are two ways of combining these behaviours, namely, *Switching and Blending* (Pan, Han, Dauber, & Law, 2007).

Switching

Switching between different behaviours is one way of producing complex navigational behaviours. With this type of selection, a pilgrim can change its behavior from one to another depending upon the perception of the environment. For example, if a pilgrim is moving towards its goal, but suddenly it faces an obstacle, it will switch its behavior from goal seeking to obstacle avoidance.

Hence, in the parallelised version of the library, switching can be done manually, or it can be automated. Adding or removing the behaviours at the host, is the manual way and is to be determined at each simulation phase. The decision of this addition or removal is made by the first layer that is, the Action Selection layer. In order to automate this process, all the relevant behaviours will run and if there is no need for the kernel to apply steering force, null value will be saved in the *steering Vectors* array.

Blending

Blending is another method to achieve complex behaviours. In this type of selection, the resultant behavior is a blend of two or more behaviours. Each of the behaviours that are combined carries a weight. This weight reflects the movement of pilgrims while navigating in the virtual environment. Hence, to take the scenario from the previous example, the pilgrim will avoid the obstacle while moving towards its goal.

For utilising the blending technique in our parallel implementation, each kernel must be assigned a weight value that is used to determine its effect on the steering of the pilgrim. These forces are then added that gives a resultant force. Before this force is applied to the pilgrim, it is limited by the maximum force value *max_force*.

The next section discusses the conversion of path following and obstacle avoidance behaviours in Steering kernels.

CONVERSION OF NAVIGATIONAL BEHAVIOURS TO STEERING KERNELS

A GPU can achieve parallelism for a portion of code that can execute independently. There are *task parallelism* and *data parallelism*. In the case of agent-based crowd simulation, the task assigned to each agent is identical. However, data and properties of each agent are different. Thus, data parallelism is more suited for this type of application. This execution model is also referred to as *Single Instruction, Multiple Threads* (SIMT) where the set of instructions is same but several different threads execute it.

OpenSteer is an open-source software, and it is developed in an object-oriented manner. The source code can be acquired from its online repository that is distributed into different folders. The **“include”** folder contains the entire header file. The **“plugins”** folder contains the sample scenarios to provide initial developers with a head start. The **“src”** folder contains the source code which provides functionality for the functions used in the library. The package also provides solution folder for three operating systems, Windows, Mac OS x and Linux. For this research, we have used the Windows operating system

The initial implementation of OpenSteer provides scenarios in the form of *plugins*. Similarly, the parallel implementation also provided plugins, but these plugins contain parallelised behaviours that measured different properties of a pilgrim. Furthermore, each thread in the CUDA based kernels is assigned to a pilgrim. All participants in a particular situation are assigned a set of behaviours in the form of kernels.

Moreover, the study proposed a parallel implementation for the path following and collision avoidance behaviours. The behaviours are converted into kernels that can be executed by multiple threads concurrently. The kernels are named as *followPathKernel* and *avoidObstaclesKernel*. These kernels are called inside the normal code by providing the grid and block configuration. A sample kernel invocation is shown in Figure 7. Calling a kernel starts with its name followed by specific characters (<<< ... >>>) to inform the compiler that this is a kernel call. These characters are followed by the list of required parameters. The kernel also requires the number of required blocks and thread.

```
void main()
{
int numThreads =  128;
int numBlocks = 8
followPathKernel<<<numBlocks, numThreads>>>(vehicleProp,
  steeringVector, direction, predictionTime);
}
```

Figure 8. Kernel invocation

Before executing the kernel, all essential data are allocated and transferred to the global memory of the GPU using c*udaMalloc* and *cudaMemcpy* functions. When the kernel is executed, the thread Id is calculated using the predefined device variable *threadIdx*. This thread id is unique and will be used to identify the agent in each scenario. This thread id is also used to indicate the properties related to the agent. This process is common for invoking both kernels from the CPU.

The kernel definition also includes some particular declaration specifier. This qualifier specifies whether the kernel is executed on the device or the host. For the kernels mentioned above, the study uses __*global*__ specifier (See Figure 9). It indicates the kernel is callable from the host and the device executes it.

```
__global__ void followPathKernel(VehicleProp *vehicleData, float3
  *steeringVectors, int *direction, float predictionTime)
{
...
}
```

Figure 9. Kernel definition

The next section an abstraction discovered in this study to facilitate future development of the parallel library.

CUDA ABSTRACT LAYER

In existing steering library, developers who want to create plugins for simulating behaviours were able to utilize their well-structured object-oriented interface. After porting the sequential steering library using CUDA and, as a result, achieving a parallel library that can make use of

GPUs. For programmers to use this accelerated library for further development, they have to look out for the following details:

- All the parallelised behaviour kernels needed to be initialised in some ways to use them in a customised plugin. To initialise it, a programmer needs to handle all the details by hand.
- Researchers will need to have CUDA development experience to deal with the API specific details that include configurations for execution and multiple function call that are all exposed to them.

In this research, to facilitate with convenient development, a C++ abstract layer was created that encloses CUDA code. The main reason for this layer was to adopt and uphold similar features that the traditional OpenSteer provided. This layer enclosed kernel initialisation details that can be used by a programmer to produce new plugins. It comprised classes that are further described in the next sections.

PlugInCUDA Class

In the serial implementation, a new plugin inherits the *PlugIn* class that handles the initialisation. In the parallel implementation, since there are additional details that need to be taken care of which includes memory allocation, transfer, and deallocation, a new abstract layer is provided. The new plugins that will use the parallel behaviours will now inherit the *PlugInCUDA* class. This new class also inherits the existing PlugIn class and in addition, to that, it also handles the extra details related to the new vehicle properties and data transfers from CPU to GPU and vice versa.

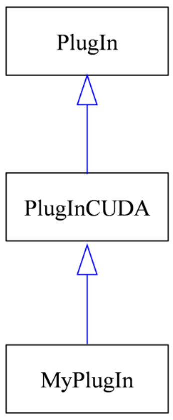

Figure 10. Class diagram for PlugInCUDA class

The next section discusses the performance results of the optimized library.

PERFORMANCE ANALYSIS AND RESULTS

Performance Metric

Frames Per Second (FPS). Frames per second or *fps* calibrates the performance of a visual application that contains a rendering of 2D or 3D environments. The OpenSteer library provides a 3D rendering of visual scenarios generated using OpenGL library. The new parallel implementation also uses the same library to render the 3D environment. This performance metrics provides the number of frames generated in one second in a simulation cycle. The range can vary greatly depending on the type of application. However, interactive frame rates are considered to be between 8-15 fps.

Execution Time of Simulation. Execution time is a key performance metric for evaluating the efficiency of an application. The execution time is inversely proportional to frames per second and expressed as:

$$execution\ time\ (ms) \propto \frac{1}{frames\ per\ second\ (fps)}$$

The equation shows that greater execution will result in fewer frames generated in one second at any given time. A high percentage of GPU utilisation will indicate a small execution time.

Speed-Up. Speedup refers to the speed of parallel application compared with the corresponding sequential application. More specifically, it shows improvement in terms of time required for execution of two different application frameworks. The term speedup was introduced in the context of Amdals's law. It can be expressed as follows:

$$S_p = \frac{T_1}{T_p}$$

Where S_p is the speedup for the parallel application, T_1 is the execution time of the sequential application, whereas T_p is the time consumed by the parallel application for completing the simulation cycle. It is necessary that while measuring the speedup, the workload must be equal. These performance metrics are necessary for measuring the efficiency of OpenSteer library. The following discussion presents the results achieved in parallelisation using the metrics mentioned previously.

An experiment on three high-performance (HPC) configuration was conducted. These HPC systems contained three different GPUs to compare their performance. The first two GPUs are from the NVIDIA® Tesla® category, C2050, and K40. The third one is from the GeForce® series, GTX 970. The results were collected by running a profiling tool provided by NVIDIA® called V*isual Profiler* (see Table 1). The primary performance metrics used for reporting the results of this research are the frames per second (fps) and execution time.

Table 1
Frames per second (FPS) for sequential and parallel implementation

No of Agents	Serial Implementation (CPU) frames per second (fps)	Parallel Implementation (GPU) frames per second (fps)		
		C2050	K40	GTX 970
500	27	83	91	101
1000	8	77	80	95
1920	6	63	70	80
2912	1	56	60	66
5920	0	36	37	45
7904	N/A	32	34	41
9920	N/A	28	29	36
19904	N/A	16	17	21
29920	N/A	12	13	18
59904	N/A	6	7	14
99904	N/A	3	5	10
199904	N/A	2	3	8
399904	N/A	1	2	3
499904	N/A	0	1	1
599904	N/A	0	1	1

The results in Table 1 shows the frames generated by the sequential and parallel implementation in a second (fps). The more the number of frames, the better the quality of the output visualisation. For a continuous video, without any noticeable delay in the frames, it should remain within the range of real-time frame rates. Eight frames per second are the minimum range of real-time frames rate. This study indicates that the fps generated by the sequential implementation cannot maintain the real-time fps for more than two thousand pilgrims. The parallel implementation shows a significant increase in performance. The high-performance configuration with NVIDIA® GTX 970 has more frames rates compared with the other two configurations for the respective number of pilgrims. The reason is that the GeForce® series is capable of handling computation as well as the visual aspect of such scientific applications. Table 2 compares different studies that have looked at the simulation of pilgrims.

Table 2
Comparison of previous studies with current implementation

Studies	Techniques	Focus	Dimensions	No of Agents	Frames Per Second (fps)
Narain et al. (2009) (Narain, Golas, Curtis, & Lin, 2009)	CFD	Tawaf	2-D	25,000	11 FPS
Zainuddin et al. (2009, 2010)	ABS	Tawaf	2-D	1000	N/A
Mulyana and Gunawan (2010)	ABS	Tawaf	2-D	150	10 FPS
Curtis et al. (2011)	ABS	Tawaf	2-D	35,000	11.5 FPS
Rahim et al. (2011)	N/A	Tawaf	3-D	500	4.3 FPS
Khan and McLeod (2012)	ABS, Cellular Automata	Tawaf	2-D	25,000	N/A
Shuaibu et al. (2013)	Mixed Mode	Tawaf	2-D	1000	N/A
Haghighati and Hassan (2013)	ABS	Tawaf	2-D	2000	N/A
Sakellariou et al. (2014)	ABS	Sa'yee	2-D	1500	N/A
Kim et al. (2015)	ABS	Tawaf	2-D	35,000	5.7 FPS
Abdur Rahman et al. (2017)	ABS	Hajj and Umrah	3-D	100,000	10 FPS

The results showed better implementation compared with previous sequential library. Crowd simulation for Tawaf and other rituals was better using this model. The parallel crowd simulation library presented in this research provided an add-on based framework through which the authorities can create multiple scenarios and event plans for different rituals of Hajj and Umrah.

CONCLUSION

This study presented a parallel implementation of navigational behaviours for agent-based crowd simulation in rituals of Hajj and Umrah which demonstrates to be a better performance. This study adds to the list of existing scientific applications that harness the high-performance capabilities of Graphics Processing Units. With a pluggable architecture of the applications, now equipped with the enhancement of high-performance computation, there is an improved program for specific event planning and management of Hajj and Umrah rituals. Experiments were conducted based on three different HPC system. The results showed robust improvements in performance for realistic crowd simulations. Encapsulation for the parallel code for further development was also provided. The research provided an abstract layer to the steering library that encapsulated the CUDA implementation. The abstraction will help developers to use improved steering behaviors.

This study was aimed at providing an optimized parallel system for large-scale agent-based crowd simulation during Tawaf ritual. There was a need for a system due to lack of computational requirements in existing sequential system to produce sufficient fps. A detailed overview of background knowledge and previous studies was followed by the design requirements of the proposed system using existing OpenSteer library. This helped in providing a new parallel model of the optimised system. The implementation of the new model had challenges. Some of the main challenges were phase traversal and appropriate data type changes. However, these problems were overcome using GPUs and CUDA toolkit for parallel implementation. The result was an optimized high-performance application which generated large-scale crowd with realistic movement performing the Tawaf ritual. The new system was able to simulate 100,000 agents in a virtual environment at 10 fps. The results discussed earlier indicates that the new system provides a better solution as compared to the previous implementation. Therefore, the main objectives of this research have been fulfilled by implementing an optimised agent-based system to simulate crowd efficiently.

ACKNOWLEDGEMENTS

This research is fully funded by the Universiti Putra Malaysia under the Fundamental Research Grant Scheme (FRGS), FRGS No: 08-02-14-1580FR.

REFERENCES

Abdelghany, A., Abdelghany, K., Mahmassani, H. S., & Al-gadhi, S. A. (2006). Microsimulation assignment model for multidirectional pedestrian movement in congested facilities. *Transportation Research Record: Journal of the Transportation Research Board,* (1939), 123–132.

AlGadhi, S., & Mahmassani, H. (1990). Modelling crowd behavior and movement: Application to Makkah pilgrimage. In *Proceedings of the 11th International Symposium on Transportation and Traffic Theory* (pp. 59–78). Yokohama. Retrieved from http://www.academia.edu/2655470/Modelling_crowd_behavior_and_movement_application_to_Makkah_pilgrimage

Cherif, F., & Chighoub, R. (2010). Crowd simulation influenced by agent's socio-psychological state. *Journal of Computing, 2*(4), 48–54. Retrieved from http://arxiv.org/abs/1004.4454

Curtis, S., Guy, S. J., Zafar, B., & Manocha, D. (2011). Virtual Tawaf: A case study in simulating the behavior of dense, heterogeneous crowds. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)* (pp. 128–135). IEEE. http://doi.org/10.1109/ICCVW.2011.6130234

Curtis, S., Guy, S. J., Zafar, B., & Manocha, D. (2013). Virtual Tawaf: A velocity-space-based solution for simulating heterogeneous behavior in dense crowds. In *Modeling, Simulation and Visual Analysis of Crowds* (Vol. 11, pp. 181–209). Springer New York. http://doi.org/10.1007/978-1-4614-8483-7_8

Gen. Auth. for Stat. Saudi Arabia. (2016). *Hajj Statistics 1437H (2016)*. Riyadh, Saudi Arabia.

Haghighati, R., & Hassan, A. (2013). Modelling the flow of crowd during Tawaf at Masjid Al-Haram. *Jurnal Mekanikal, 36*(1), 2–18.

Helbing, D., & Farkas, I. (2002). Simulation of pedestrian crowds in normal and evacuation situations. *Pedestrian and Evacuation Dynamics, 21*(2), 21–58. Retrieved from http://www.researchgate.net/publication/224010870_Simulation_of_pedestrian_crowds_in_normal_and_evacuation_situations/file/d912f50eb0d9bb6224.pdf

Hoogendoorn, S. P., & Bovy, P. H. L. (2001). Generic gas-kinetic traffic: Systems modeling with applications to vehicular traffic flow. *Transportation Research Part B: Methodological, 35*(4), 317–336. http://doi.org/10.1016/S0191-2615(99)00053-3

Jennings, N. R. (2000). On agent-based software engineering. *Artificial Intelligence, 117*(2), 277–296. http://doi.org/10.1016/S0004-3702(99)00107-1

Khan, I., & McLeod, R. (2012). Managing hajj crowd complexity: Superior throughput, satisfaction, health, and safety. *Kuwait Chapter of Arabian Journal of Business and Management Review, 2*(4), 45–59. Retrieved from http://www.arabianjbmr.com/pdfs/KD_VOL_2_4/4.pdf

Kim, S., Guy, S. J., Hillesland, K., Zafar, B., Gutub, A. A. A., & Manocha, D. (2015). Velocity-based modeling of physical interactions in dense crowds. *The Visual Computer, 31*(5), 541–555. http://doi.org/10.1007/s00371-014-0946-1

Klüpfel, H. (2007). The simulation of crowds at very large events. In *Traffic and Granular Flow'05* (pp. 341–346). Berlin, Heidelberg: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-47641-2_30

Mulyana, W. W., & Gunawan, T. S. (2010). Hajj crowd simulation based on intelligent agent. In *2010 International Conference onComputer and Communication Engineering (ICCCE)* (pp. 1-4). IEEE. http://doi.org/10.1109/ICCCE.2010.5556818

Narain, R., Golas, A., Curtis, S., & Lin, M. C. (2009). Aggregate dynamics for dense crowd simulation. *ACM Transactions on Graphics, 28*(5), 122. http://doi.org/10.1145/1618452.1618468

Pan, X., Han, C. S., Dauber, K., & Law, K. H. (2007). A multi-agent based framework for the simulation of human and social behaviors during emergency evacuations. *Ai and Society, 22*(2), 113–132. http://doi.org/10.1007/s00146-007-0126-1

Rahim, M. S. M., Fata, A. Z. A., Basori, A. H., Rosman, A. S., Nizar, T. J., & Yusof, F. W. M. (2011). Development of 3D Tawaf simulation for hajj training application using virtual environment. In *Visual Informatics: Sustaining Research and Innovations: Second International Visual Informatics Conference, IVIC 2011, Selangor, Malaysia, November 9-11, 2011, Proceedings, Part I* (pp. 67–76). Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-642-25191-7_8

Rahman, A., Hamid, N. A. W. A., Rahiman, A. R., & Zafar, B. (2015). Towards accelerated agent-based crowd simulation for Hajj and Umrah. In *2015 International Symposium on Agents, Multi-Agent Systems and Robotics (ISAMSR)* (pp. 65–70). IEEE. http://doi.org/10.1109/ISAMSR.2015.7379132

Reynolds, C. (2006). Big fast crowds on PS3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames - Sandbox '06* (pp. 113–121). New York, New York, USA: ACM Press. http://doi.org/10.1145/1183316.1183333

Sakellariou, I., Kurdi, O., Gheorghe, M., Romano, D., Kefalas, P., Ipate, F., & Niculescu, I. (2014). Crowd formal modelling and simulation: The Sa'yee ritual. In *2014 14th UK Workshop on Computational Intelligence (UKCI)* (pp. 1–8). IEEE. http://doi.org/10.1109/UKCI.2014.6930176

Salamati, P., & Rahimi-Movaghar, V. (2016). Hajj stampede in Mina, 2015: Need for intervention. *Archives of Trauma Research, 5*(5), 1. http://doi.org/10.5812/atr.36308

Sarmady, S., Haron, F., & Talib, A. Z. (2011). A cellular automata model for circular movements of pedestrians during Tawaf. *Simulation Modelling Practice and Theory, 19*(3), 969–985. http://doi.org/10.1016/j.simpat.2010.12.004

Shuaibu, A. N., Faye, I., Malik, A. S., & Talal, M. (2014). Collision avoidance path for pedestrian agent performing Tawaf. In T. Herawan, M. M. Deris, & J. Abawajy (Eds.), *Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013)* (Vol. 285, pp. 361–368). Singapore: Springer Singapore. http://doi.org/10.1007/978-981-4585-18-7

Shuaibu, N. A., Faye, I., Simsim, M. T., & Malik, A. S. (2013). Spiral path simulation of pedestrian flow during Tawaf. *IEEE ICSIPA 2013 - IEEE International Conference on Signal and Image Processing Applications* (pp. 241–245). IEEE. http://doi.org/10.1109/ICSIPA.2013.6708011

Zafar, B. (2011). *Analysis of the Mataf - Ramadan 1432 AH. Technical report.* Makkah, Saudi Arabia.

Zainuddin, Z., Thinakaran, K., & Abu-Sulyman, I. M. (2009). Simulating the circumambulation of the Ka'aba using SimWalk. *European Journal of Scientific Research, 38*(3), 454–464.

Zainuddin, Z., Thinakaran, K., & Shuaib, M. (2010). Simulation of the pedestrian flow in the Tawaf area using the social force model. *World Academy of Science, Engineering and Technology, 48,* 908–913.