The Virtual Marathon: Parallel Computing Supports Crowd Simulations

Erdal Yilmaz, Veysi Isler, and Yasemin Yardimci Çetin ▪ *Middle East Technical University*

To be realistic, an urban model must include appropriate numbers of pedestrians, vehicles, and other dynamic entities. Using a parallel-computing architecture, researchers simulated a marathon with more than a million participants. To simulate participant behavior, they used fuzzy logic on a GPU to perform millions of inferences in real time.

Although crowds are a part of daily metropolitan life, massive crowds typically appear only at special events, such as concerts, political rallies, or sporting events. A marathon is one of the largest events, including a huge population of runners as well as many spectators. For example, at the annual New York City Marathon, nearly 40,000 runners participate, and one million people watch them from sidewalks.

Many other cities worldwide also host well-known marathons. Figure 1a shows a massive crowd of runners on the Bosporus Bridge in Istanbul during the Intercontinental Istanbul Eurasia Marathon. At this annual event, thousands of people attempt an intercontinental course between Asia and Europe. This event, along with the lack of crowd simulation studies related to marathons, inspired us to investigate marathon simulation. (For a look at related work in crowd simulation, see the sidebar.) Figure 1b is a screenshot from the resulting simulation, showing thousands of virtual athletes running over the Bosporus Bridge.

To achieve real-time, lifelike performance, such a simulation requires effective use of hardware as well as well-known computer graphics algorithms such as LOD (level of detail) and frustum and occlusion culling. To meet the real-time requirements, we used parallel processing on a GPU, employing CUDA (originally Compute Unified Device Architecture). A GPU opens the door to more realism and better frame rates because it absorbs the calculation overload from a CPU. We also exploited parallel processing to create characters with more realistic behavior, through fuzzy logic. Using fuzzy logic for crowd simulation isn't new, but programming a GPU with CUDA to perform millions of fuzzy inferences in real time is.

CUDA

In 2007, Nvidia released its CUDA parallel-processing architecture for next-generation GPUs, letting programmers use C. CUDA introduces a GPU as a coprocessor to meet the requirements of power-demanding operations that a CPU couldn't handle, in addition to graphics and rendering tasks. Because modern GPUs have many cores, they offer large performance benefits for parallel processing.

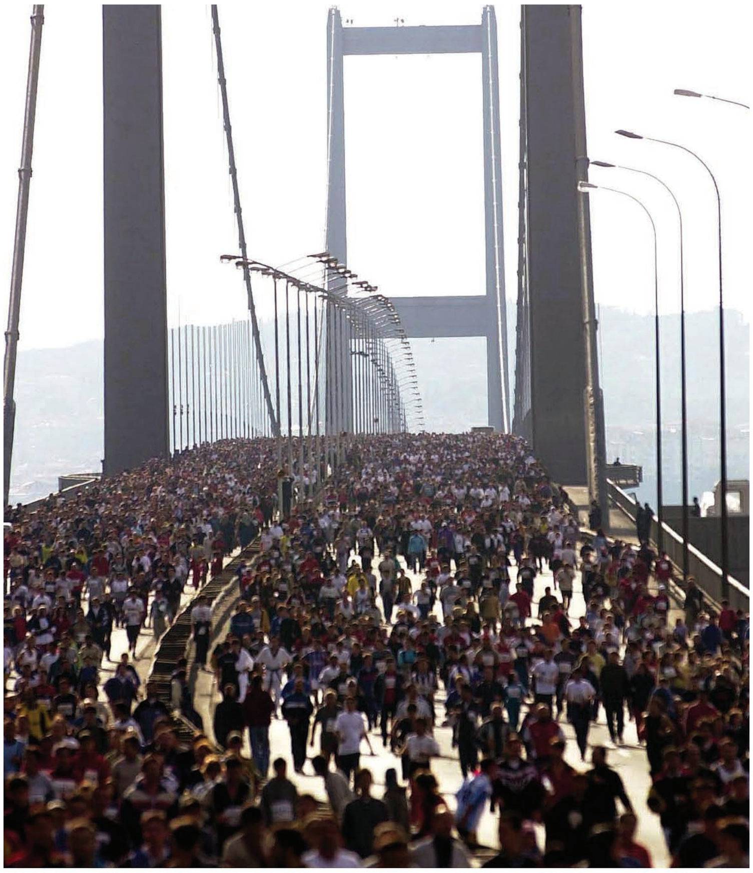
Recently, researchers demonstrated significantly increased speedup after adapting existing CPU-oriented algorithms to parallel processing with CUDA. Lars Nyland and his colleagues achieved 50× speedup on an *N*-body simulation in which every physical body interacts with the others.[1] This speedup occurred in a highly optimized CPU implementation, which produced 200× speedup compared to an average CPU implementation of the same problem. Several other studies report similar speedup in different domains.[2,3] These promising achievements encouraged us to use CUDA in our marathon simulation.

Implementation

Our simulation involved more than one million virtual people (32,768 runners and 1,015,808 spectators). We also constructed a virtual city model containing thousands of buildings and city furniture models (traffic lights, street lamps, and so on). To compile the simulation, we used Visual C++ and CUDA. We used OpenGL graphics programming. The simulation ran on a PC with an Intel Core 2 Quad CPU, 3 Gbytes of RAM, and an Nvidia GeForce GTX280 GPU (1 Gbyte of video RAM and 240 stream processors). For higher screen resolution, we used a multimonitor setup connected with a Matrox TripleHead2Go card. Three connected 19-inch LCD monitors produced 3,840 × 1,024 pixel resolution (see Figure 2a). An upper monitor served as extra workspace and wasn't used for rendering. As Figure 2b shows, we can increase resolution further by enhancing the multimonitor setup.

We used commercially available human models with 4,000 to 6,000 polygons and high-resolution texture maps. We used rigged models so that we could animate them easily by using motion-capture data. We applied different texture maps for human-model variation. To create different body shapes, we scaled each model along three axes (see Figure 3). To increase rendering performance, we used four LODs for all 3D models; each LOD had 50 percent of the previous level's detail. We produced the LODs offline using Autodesk's 3D Studio Max 8, which has a polygon reduction feature that generates low-polygon instances. The polygon constructs were imported to and manipulated within the simulation system.

Instead of polygonal models, we could have used image-based (*impostor*—an impostor is a 2D image produced by rendering a 3D complex object), point-based, or hybrid rendering techniques for each LOD. Using impostors would require additional preprocessing to prepare texture maps for many animation sequences from various view angles. When the crowd depth is high, point-based rendering lets you use low-detail representation for far-away objects. However, the marathon scene's side view couldn't benefit from the mesh reduction because of lack of depth. Detailed study of LOD techniques appear elsewhere.[4]


(a)

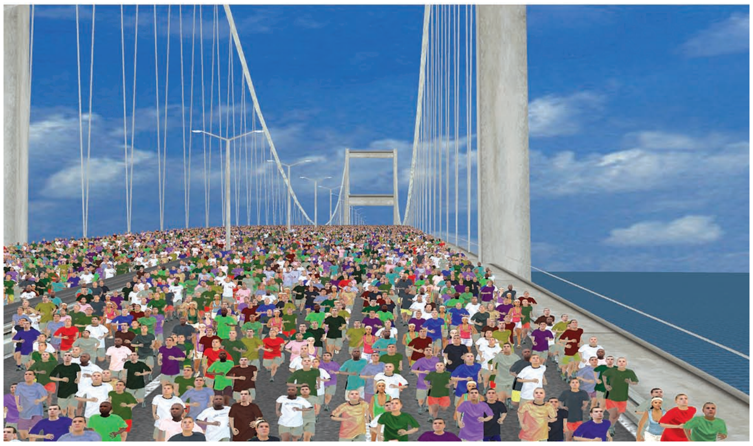

(b)

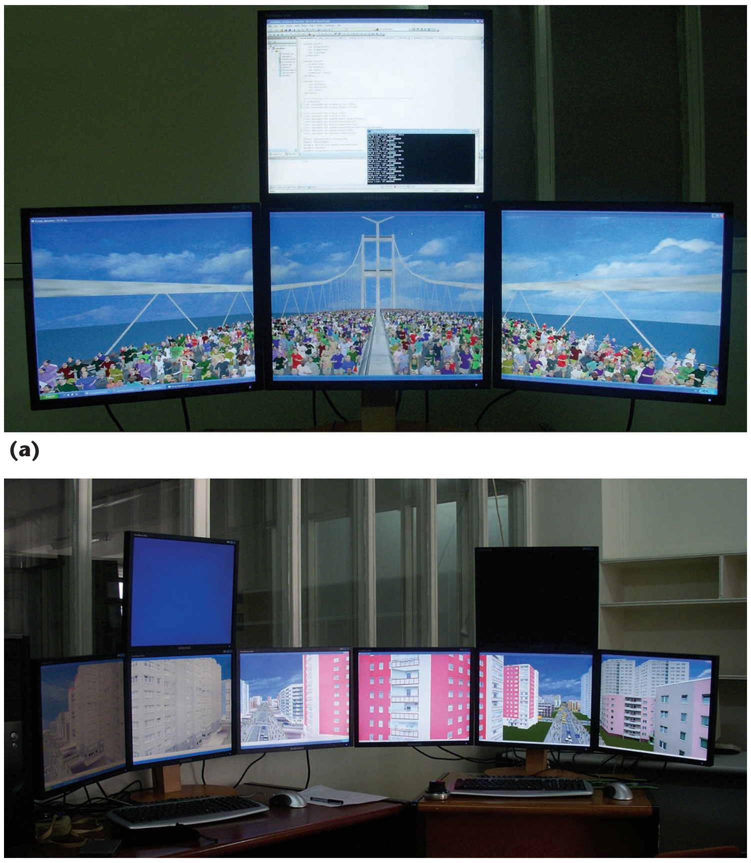
Figure 1. Marathon crowds: (a) a photo of a crowd of runners on Istanbul's Bosporus Bridge and (b) a screenshot of a virtual-marathon simulation. Previous research on crowd simulation hasn't dealt with marathons. (Figure 1a source: Istanbul Metropolitan Municipality; used with permission.)

Application Workflow

Figure 4 shows the application workflow. The *application initialization* step constructs the virtual city model and assigns personal and physical values to each individual. It also transfers fuzzy sets and knowledge base data to the GPU for the virtual people's reasoning process (AI).

Next, the main simulation loop starts. The *start simulation* step sets the view frustum parameters and timers and updates several global variables. This step's average runtime doesn't significantly affect simulation loop time, so we don't include the details.

Efficient updating of the crowd is this study's most important part. The *update individuals* step


(a)

(b)

Figure 2. Multimonitor setups for the virtual marathon. (a) The basic setup consisted of three connected 19-inch LCD monitors, which produced 3,840 × 1,024 pixel resolution. (b) An enhanced multimonitor setup provides increased resolution.

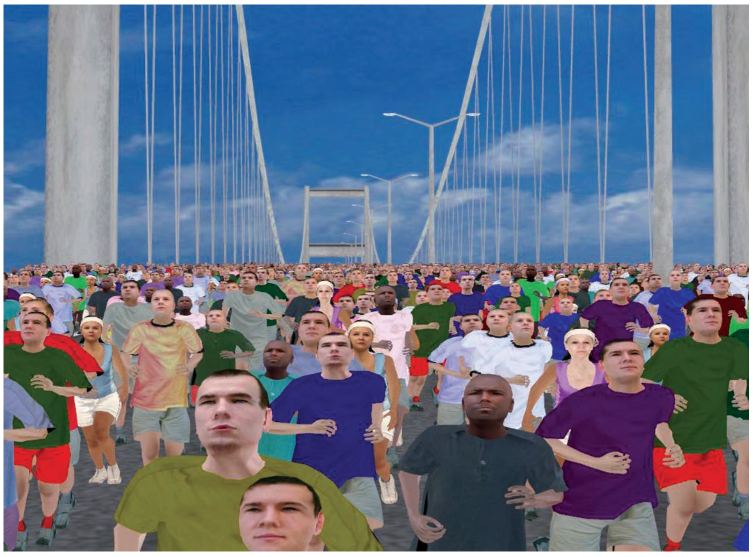


Figure 3. A close-up view of the runners. We achieve model variety by scaling the models with random values and associating different texture maps with the human characters.

refreshes all information for each virtual human, even those outside the view frustum or far from the virtual-camera center. This step performs all operations on a GPU using CUDA. Parallel processing and the GPU's many cores produce huge computational power, enough to update massive crowds. During this step, virtual characters' dynamic parameters such as position, direction, speed, and current feelings transfer from the CPU (host) to the GPU (device), together with view frustum planes and camera positions. This step handles each individual on a separate thread. Depending on the GPU configuration, thousands of virtual humans can be processed in parallel.

The GPU's first job is to calculate a processed entity's LOD and visibility by calculating the distance between the character and camera position to determine whether the entity is in the view frustum (is visible). This task is one of the most common in real-time computer graphics applications, including crowd simulation. CUDA makes it trivial.

Next, AI processing determines agent behavior. We describe this process in the next subsection.

Then, the navigation starts, and the system repositions the characters. We use predefined paths; we haven't yet implemented a path-finding algorithm. Similarly, we haven't implemented collision prevention on the GPU. These are major considerations for this research's future. During navigation, the system calculates current positions on the basis of parameters such as speed and direction and updates these values if necessary. In each simulation time step, the system calculates the virtual characters' vertical positioning (see Figure 5) to prevent the sink-or-raise problem, which would cause virtual characters to be drawn below or above the ground surface.

The *LOD0 retouch* step covers very limited collision detection performed on the CPU, which only covers individuals close to the viewpoint. Our current algorithm requires performing collision tests between all characters inside specified spatial, neighboring cells. In this step, the CPU can also be used for computationally intensive exceptional threads, which could cause a delay on the GPU due to the GPU's SIMT (single instruction, multiple threads) architecture.

Finally, the *rendering* step renders the virtual characters in the view frustum. In this step, the GPU handles the graphics API, not CUDA. After rendering, the simulation loop restarts.

Depending on the scene's complexity, we achieved between 10 to 30 frames per second (fps). When we simulated more than one million humans, the average GPU update time was approximately 60 milliseconds. Rendering required 25 milliseconds for a scene with nearly 10,000 virtual humans (represented with 3D geometric models). Other steps required approximately 1 millisecond. In such a case, one simulation loop finished in approximately 85 milliseconds, which corresponds to 11 to 12 fps.

Fuzzy Logic with CUDA

Fuzzy logic provided a way to make a decision based on a membership value ranging from 0 to 1 rather than true or false. In this way, fuzzy logic helped us produce distinct behaviors.

Our AI implementation updated each individual's status without considering that individual's visibility. Each simulation loop involved several millions of fuzzy logic inference operations, thanks to the GPU's parallel-processing capability. We implemented fuzzy logic inference functions from scratch because existing fuzzy logic libraries aren't designed for CUDA. When implementing those functions on the GPU, we followed Penny Baillie-de Byl's work.[5]

The fuzzy logic inference updates individuals' feelings or reasoning mechanisms and increases the simulation's realism. We can observe this in how the virtual spectators react during the marathon; they stand on sidewalks and cheer as runners pass. The runners' order and the spectators' excitement level determine the spectators' cheering level and style. For example, spectators cheer more for front-line runners (see Figure 6). This inference also evaluates spectator support for a specific runner such as a friend or relative. After runners pass by, spectators get bored and the inference output changes dynamically. Similarly, runners decide to increase or decrease their pace depending on their goals, surrounding parameters, and physical condition. Most inference inputs are dynamic and might change during simulation. However, some are fixed, such as each runner's goal (to break a course record, break a personal record, finish the race, or have fun).

We implemented a four-step Mamdani-style fuzzy inference: fuzzification, rule evaluation, aggregation, and defuzzification.[5] All the fuzzy sets and rules (the knowledge base) are fixed and passed from host to device during initialization. In each simulation frame, the main CUDA kernel function calls the `evaluateRule` function for each rule in the fuzzy inference. This function produces a new fuzzy subset. After rule evaluation, the union of fuzzy subsets produces a new fuzzy set.[5] Finally, the main kernel function calls the `getCentroid` function to compute a scalar value for modeling a virtual character's individual behavior.

For CUDA implementation, we use these fuzzy inference functions:

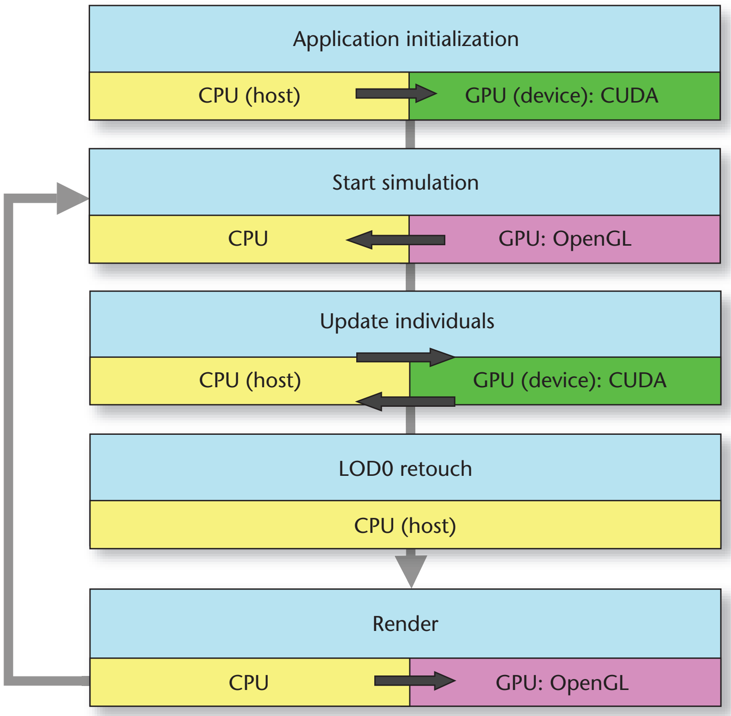


Figure 4. The application workflow. The GPU provides rendering and computation during different simulation steps.

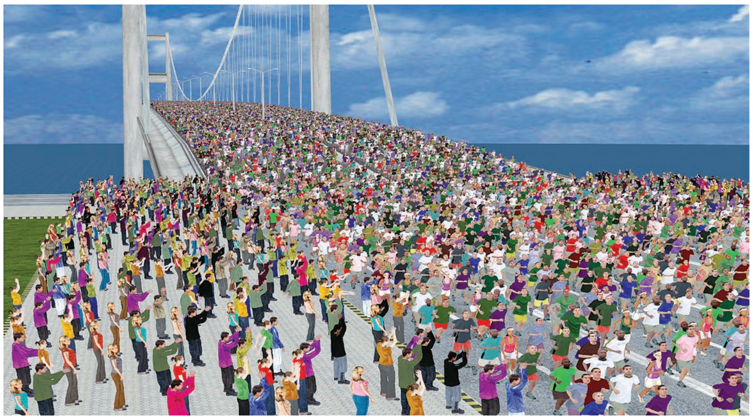


Figure 5. Vertical positioning of characters. The GPU precisely calculates the contact point between virtual people and roads or sidewalks.

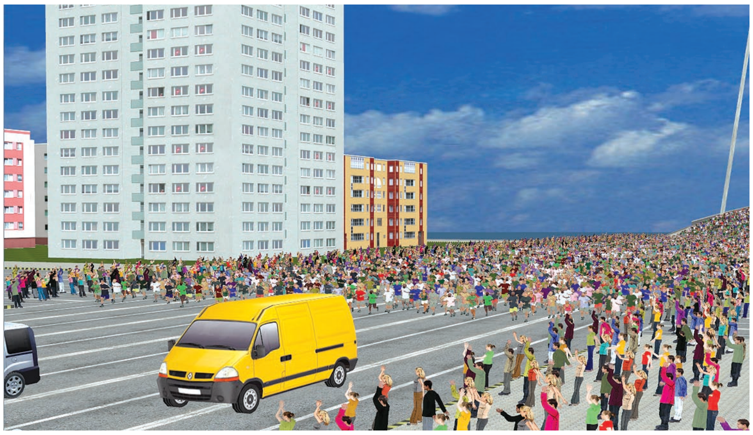


Figure 6. Spectators cheer for the front-line runners. The simulation determines spectator behavior individually via fuzzy inferences.

Related Work in Crowd Simulation

Here we look at two main research areas: rendering and simulation with parallel processing.

Rendering Large Crowds

The number of virtual people in real-time applications has increased significantly since the late 1990s, owing to enormous improvements in graphics hardware, performance-increasing graphics algorithms, and solution-oriented rendering techniques. Although conservative frustum- and occlusion-culling techniques and a low level of detail (LOD) help decrease a rendering system's load, they can't by themselves achieve interactive frame rates in massive crowd simulation applications. Image-based rendering techniques, which are a life jacket for crowded virtual environments, simply represent virtual characters with fewer polygons, mostly quads, instead of high-polygon 3D models. By using this image-based approach and Nvidia's 64-Mbyte GeForce GTS2 card, Franco Tecchia and his colleagues visualized a village of 10,000 people at approximately 20 frames per second.[1]

Simon Dobbyn and his colleagues offered a novel hybrid rendering technique by using polygonal models and impostors (an impostor is a 2D image produced by rendering a 3D complex object) derived from these models to overcome degraded image quality at close viewing distances.[2] They named their impostors *geopostors* because their algorithm produces these impostors directly from a 3D geometric model. This approach also helped them overcome the pop-up problem that occurs when the computer switches between model representations. If a noticeable difference exists between these models (size, color, animation phase, and so on), it might be perceptual.[2] In addition, they used several GPU facilities to increase visual quality and virtual-actor variety. They demonstrated up to 30,000 virtual people at interactive frame rates using a GeForce 4 Ti4600 3D card with 128 Mbytes of memory.

In another study, Ladislav Kavan and his colleagues minimized texture-memory consumption significantly by introducing the *polypostor*, which is a multipolygon impostor rather than a single quad.[3] This structure still represents a virtual character with few polygons and uses a single, smaller texture map. This approach realizes animation by changing vertex geometry. Not surprisingly, it performs similarly to quad-based impostors because the graphics pipeline can manage the number of polygons. Using polypostors consisting of 90 polygons, Kavan and his colleagues showed they could render up to 120,000 virtual people. However, the actual number of rendered entities was lower owing to frustum and occlusion culling.

Erik Millán and Isaac Rudomín further increased the number of virtual people by exploiting GPU processing functionalities.[4] They achieved interactive frame rates for as many as 250,000 impostors and lower frame rates for more than one million virtual people. They focused mainly on crowd density, using simple navigation, animation, and behavior models. Similarly to Dobbyn and his colleagues, they employed 3D geometric models to visualize characters close to the viewpoint. To accelerate this process, they applied pseudo-instancing, which uses a single API draw call to improve rendering performance.

Parallel Computing and Virtual Crowds

The research we just described used simple models to address behavior- and navigation-related issues. However, the demand for realism involves not only lifelike graphics but also artificial intelligence, smooth navigation, and physical modeling. Lifelike real-time virtual environments with massive crowds require more processing power than a commodity PC can provide. Parallel computing might help meet the requirements of computing-intensive opera-

- `getMembershipDegree` yields the degree of membership in a fuzzy set. For computational performance and simplicity, we used only linear-fit functions.
- `evaluateRule` performs a fuzzy operation on given fuzzy sets to compute a scalar value to clip the output fuzzy set. We implemented only fuzzy `AND` and fuzzy `OR` operations.
- `clipFuzzySet` makes a new fuzzy subset by clipping the output fuzzy set.
- `aggregateFuzzySets` creates a new fuzzy set by combining clipped fuzzy sets.
- `getCentroid` calculates the center of aggregated fuzzy sets.

Figure 7 shows a simple CUDA kernel implementation workflow of a fuzzy knowledge base.

Instead of using fuzzy logic for behavioral modeling, we could have used finite state machines (FSMs). Isaac Rudomín and his colleagues used GPUs to simulate agent behavior in crowd simulation via FSMs that they implemented as fragment shaders using GLSL (OpenGL Shading Language).[6] Specifically, they used FSMs with texture maps as easily accessible lookup tables. CUDA eliminates the indirect use of shaders and texture maps and provides a simpler coding environment than GLSL. Fuzzy logic provides a higher degree of variety than deterministic FSMs but requires more coding effort to implement.

For more variety, researchers have introduced probabilistic FSMs.[5] These constructs still involve a finite number of states but determine transitions between states according to given transition probtions in crowd simulations. The approaches we report in this section don't benefit from basic load-minimizing techniques such as frustum and occlusion culling or LOD. They also handle every agent the same way, independent of visibility or distance.

Michael Quinn and his colleagues accelerated the simulation of pedestrian movement on the basis of a social-powers model using a cluster of 11 personal computers and an MPI (message passing interface) library.[5] The library is a software utility that handles parallel-processing tasks, including data transfer between CPUs. To meet real-time constraints, they used a PC cluster organized in a manager-worker architecture. The manager PC handles communication with the worker PCs. It collects each individual's current position after updating the cycle and passes this information to the rendering engine. This architecture minimizes network traffic by eliminating worker intercommunication. Quinn and his colleagues observed a linear performance increase when adding more PCs.

In a similar study, Anthony Steed and Roula Abou-Haidar focused on dynamic allocation of regions for crowd distribution, using spatial-partitioning algorithms.[6] With nonuniform distributions, parallel processing might not work as intended, especially in real-time applications where synchronization is a major issue. If crowd density in similar-sized regions differs significantly, load-balancing precautions are necessary.

Bo Zhou and Suiping Zhou partitioned flock simulation on a PC cluster with MPI to simplify $O(n^2)$ complexity and increase the number of entities.[7] They examined different network topologies and reported that *near-neighbor communication*, in which a PC is connected only to PCs on either side, is the best. They also demonstrated that dynamic load balancing increases performance when used infrequently.

Finally, Craig Reynolds used parallel processing for fish simulation on the PlayStation 3, which has one PowerPC processor and seven Synergistic Processing Units.[8] Reynolds modeled crowds as interacting particle systems, with each agent checking the rest for interaction ($O(n^2)$ complexity). This approach was easy to implement with only a few virtual actors but required more-advanced algorithms for more than several thousand actors. To overcome this problem, Reynolds used spatial hashing.

References

1. F. Tecchia et al., "Visualizing Crowds in Real-Time," *Proc. Symp. Interactive 3D Graphics and Games*, ACM Press, 2005, pp. 95–102.
2. S. Dobbyn et al., "Geopostors: A Real-Time Geometry/Imposter Crowd Rendering System," *Computer Graphics Forum*, vol. 21, no. 4, 2002, pp. 753–765.
3. L. Kavan et al., "Polypostors: 2D Polygonal Impostors for 3D Crowds," *Proc. Symp. Interactive 3D Graphics and Games,* ACM Press, 2008, pp. 149–155.
4. E. Millán and I. Rudomín, "Impostors, Pseudo-Instancing and Image Maps for GPU Crowd Rendering," *Int'l J. Virtual Reality,* vol. 6, no. 1, 2007, pp. 35–44.
5. M.J. Quinn et al., "Parallel Implementation of the Social Forces Model," *Proc. 2nd Int'l Conf. Pedestrian and Evacuation Dynamics*, School of Computing and Mathematical Sciences, Univ. of Greenwich, 2003, pp. 63–74.
6. A. Steed and R. Abou-Haidar, "Partitioning Crowded Virtual Environments," *Proc. Symp. Virtual Reality Software and Technology,* ACM Press, 2003, pp. 7–14.
7. B. Zhou and S. Zhou, "Parallel Simulation of Group Behaviors," *Proc. 2004 Winter Simulation Conf.,* IEEE CS Press, 2004, pp. 364–370.
8. C. Reynolds, "Big Fast Crowds on PS3," *Proc. Siggraph Symp. Videogames,* ACM Press, 2006, pp. 113–121.

```
kernel (main CUDA kernel function)
  For each fuzzy rule perform rule evaluation (call evaluateRule)
  evaluateRule function
    For each fuzzy set perform fuzzification (call getMembershipDegree)
    Evaluate rule using fuzzy operator (AND/OR)
    Make new fuzzy subset (call clipFuzzySet)
  For each fuzzy subset perform aggregation (call aggregateFuzzySets)
  Perform defuzzification (call getCentroid)
  Set individual behavior
```

Figure 7. A simple CUDA kernel implementation workflow of a fuzzy knowledge base. The simulation carries out AI processing using the given functions in parallel on the GPU, on the basis of CUDA.

abilities. Fuzzy logic has no significant computational advantage over probabilistic FSM. However, it enables an entity to be in multiple states at any time and thus is more suitable for modeling complex human behavior and interactions. We plan to expand our research to include other dimensions of city life. Fuzzy logic provides a natural language for translating human experience into a knowledge base for such complex environment simulation.

CPU and GPU Comparison

We compared the CPU and GPU performance to see GPU parallel processing's potential for massive-

Table 1. CPU and GPU processing times for updates.

Number of people	Processing time (ms)			
	Low-cost model		High-cost model	
	CPU	GPU	CPU	GPU
32,768	46.25	3.10	198.11	3.16
65,536	90.64	4.32	394.39	4.36
131,072	179.36	8.59	786.24	8.76
262,144	356.48	15.52	1,573.16	15.60
524,288	711.28	30.89	3,119.36	31.29
1,048,576	1,420.64	59.86	6,282.52	61.20

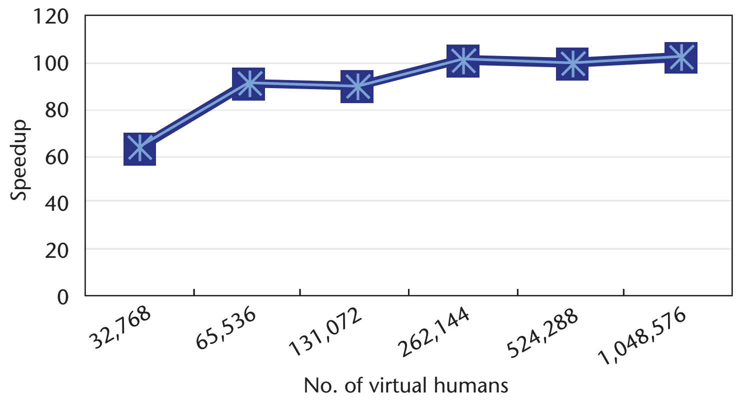


Figure 8. GPU speedup for the high-cost model. We achieved nearly 100× speedup on a GPU using CUDA.

crowd simulation. We used two models: the low-cost model included one fuzzy inference, and the high-cost model included four fuzzy inferences and used a more precise frustum-culling approach. Consequently, it required nearly five times more computation. Table 1 shows the results.

For the high-cost model, the GPU calculated virtual-character navigation and reasoning almost 100 times faster than the CPU (see Figure 8). The GPU performed better as the number of entities and computational cost increased. CPU processing time was exactly linear, whereas there was no significant change for the GPU owing to the natural result of parallel processing. As Table 1 shows, the GPU processing times for the two models aren't significantly different, despite a 5× computational difference. This difference was due to data transfer between the host and the device or vice versa. Because data transfer costs much more than GPU processing time, the computational difference of a few hundred flops is negligible. This conclusion is valid only for cases similar to our study.

One of the worst scenarios for parallel processing of crowd simulations would be individuals who require extremely expensive computation. To see what happens when some people require more computation than others, we identified 10 people requiring 1,000 times more computation than the rest of the crowd. This new scenario created no significant difference for the CPU because it was the same as adding 10,000 more people. On the GPU, however, all the other threads had to wait for the longest-running one, resulting in degraded performance. With a small number of computationally demanding people (such as 10 out of 32,768), the GPU performed worse than the CPU. As this ratio decreased (for example, 10 of 1,048,576), the delay became negligible.

To achieve higher speedup, it's important to process a group of people with similar computational requirements in a single GPU warp. So, we assigned 10 successive thread indices to the previously identified 10 people. As a result, GPU performance increased significantly because we handled those 10 individuals in a single warp. Depending on the number of such entities, we could filter them to minimize idle time on parallel-processing resources. We could also process them using the CPU. This test demonstrated the importance of considering individual computational cost in the simulation phase when entities are grouped. This grouping becomes crucial if a significant difference exists between the computational requirements of individuals in the entire crowd, unless spending resources on the classification task isn't viable. In light of these findings, we filtered the LOD0 people (those close to the viewpoint) and performed their collision detection on the CPU because the algorithm required more processing power.

Results and Discussion

When using GPU parallel processing for crowd simulation, we must minimize idle time to increase effectiveness. The perfect case would be to have exact multiples of GPU threads, with each thread having the same computational cost. In this case, all threads would complete tasks simultaneously with no idle time. Recreating this case in crowd simulations isn't easy because various factors in navigation (path finding and collision prevention) and reasoning (individual AI and group and personal behavior) can cause different computation times.

Another issue is the possibility of increasing the frame rate by using more GPUs. Scaling in multi-GPU applications is almost 100 percent. The CUDA software development kit contains several multi-GPU samples that demonstrate this scaling. (However, Nvidia supports multi-GPU only on the same cards.) Multi-GPU processing increases computational power significantly. Currently, you can connect eight CUDA devices on one PC board.

As we mentioned earlier, CUDA lets us program GPU functionality in C. Recently, Qiming Hou and his colleagues introduced the BSGP (Bulk-Synchronous GPU Programming) language, which provides an easy coding environment for general-purpose computations on a GPU as well.[7] They reported that BSGP performs comparably to CUDA but requires less coding effort. CUDA's increasing popularity and the introduction of such new programming environments simplifies porting real-time crowd simulations to a GPU.

For future research, our priority is to perform collision detection on the GPU instead of the CPU. Then, every operation regarding crowd simulation would be on a GPU using CUDA. We'd also like to add more GPUs and use three or four of these setups to develop a hybrid parallel-processing system comprising heterogeneous hardware with multiple CPUs, GPUs, and monitors. Such a system should possess enormous processing power. It should be able to simulate and render more-complex scenes with an exact marathon course, including start and finish lines, refreshment stations, and entities such as bikers, officials, and volunteers.

References

1. L. Nyland et al., "Fast *N*-Body Simulation with CUDA," *GPU Gems 3,* Addison-Wesley, 2008, pp. 677–695.
2. M. Silberstein et al., "Efficient Computation of Sum-Products on GPUs through Software-Managed Cache," *Proc. 22nd Ann. Int'l Conf. Supercomputing* (SC 08), ACM Press, 2008, pp. 309–318.
3. L. Howes and D. Thomas, "Efficient Random Number Generation and Application Using CUDA," *GPU Gems 3,* Addison-Wesley, 2008, pp. 805–830.
4. E. Millan and I. Rudomin, "Impostors, Pseudo-Instancing and Image Maps for GPU Crowd Rendering," *Int'l J. Virtual Reality*, vol. 6, no. 1, 2007, pp. 35–44.
5. P. Baillie-de Byl, *Programming Believable Characters for Computer Games,* Charles River Media, 2004, pp. 212–230.
6. I. Rudomín, E. Millán, and B. Hernández, "Fragment Shaders for Agent Animation Using Finite State Machines," *Simulation Modeling Practice and Theory*, vol. 13, no. 8, 2005, pp. 741–751.
7. Q. Hou et al., "BSGP: Bulk-Synchronous GPU Programming," *ACM Trans. Graphics*, vol. 27, no. 3, 2008, article 19.

Erdal Yilmaz *is a PhD candidate at Middle East Technical University's Informatics Institute. His research interests include computer graphics, video games, artificial intelligence, geographic information systems, and their applications in the defense industry. Yilmaz has an MSc in information systems from Middle East Technical University's Informatics Institute. Contact him at erdal@ii.metu.edu.tr.*

Veysi Isler *is a faculty member in Middle East Technical University's Department of Computer Engineering. He's also the director of the university's Modeling and Simulation Research and Development Center. His research interests include rendering, visualization, game technology, and parallel rendering algorithms. Isler has a PhD in computer engineering. Contact him at veysi@metu.edu.tr.*

Yasemin Yardimci Çetin *is a professor at Middle East Technical University's Informatics Institute. Her research interests include image registration, computer graphics, and target detection. She's the chair of the IEEE Signal Processing Society and the interim chair of the IEEE Aerospace and Electronic Systems Society Chapters of Turkey. Çetin has a PhD in electrical engineering from Vanderbilt University. Contact her at yardimy@ii.metu.edu.tr.*