RoboCup Rescue Simulator on Graphics Processing Units

Prashant Sethia and Kamalakar Karlapalem

Abstract. In this paper, we present the utility of porting the code for RoboCup Rescue Simulator with the same architecture on top of Graphics Processing Units. We use Compute Unified Device Architecture (CUDA) that leverages the parallel compute engine in nVidia graphics processing units (GPUs) to solve the computational problems related with the simulator. We find that the SIMT (Single Instruction Multiple Thread) architecture of GPUs is coherent to the nature of rescue simulations. We model agents as light-weight GPU threads (maximum number of threads possible is 2^{41} in CUDA) which execute in parallel on the multi-core GPU to do computations in a fraction of the time required by the CPU. Further, execution on GPU provides an opportunity for parallel execution of a large number of agents which allows different events to occur simultaneously in the simulated agent universe giving more realistic results in each snapshot. Most importantly we are able to increase the number of agents in the rescue simulations from a few thousands (in the current simulator) to a few millions.

Key words: RoboCup Rescue Simulator, GPU, CUDA, Multi-core

1 Introduction

Developing tools for disaster-rescue simulations has been an active area of research in the last decade, with emphasis being laid on different aspects - architecture, scalability, efficiency, robustness, effectiveness in modelling - one such tool being RoboCup Rescue Simulator [5]. The current version of the simulator utilizes Central Processing Units (CPU), be it a single CPU with single core or a grid of processing units with multiple cores. In CPU based simulations performance is often raised by increasing the number of cores. Further, the current simulator uses Java objects for modelling agents and the geographical informations which require large amounts of main memory, thus limiting the maximum number of civilian and rescue agents to a few thousands. Moreover, in simultaneous multi-threading on CPUs the number of concurrent threads in a single core is limited to two and context switching between threads is heavy, thus limiting the number of parallel threads to a few hundreds. Consequently, the rescue simulator executes the different agents in cycles in a limited number of threads, rather than executing each agent in parallel in a different thread.

The possibility of porting the simulator code on GPUs has not been studied till now. We coded a dummy rescue simulator, having the same architecture as the current version of RoboCup Rescue Simulator, for execution on CUDA driven nVidia GPUs. When executing on GPUs, we were able to scale up the total number of agents in the rescue simulation to ten million. Further, the number of concurrent threads on a single Tesla T10 GPU is 30,720. Thus, the amount of concurrency provided by the GPUs is significantly higher than the CPUs. This factor is a boon

to rescue simulations because we can get rid of cycle based simulation and pave way for concurrent, parallel simulations with each agent acting in a separate thread, and events occuring simultaneously in parallel as they are supposed to be in real life. Increased number of agents and concurrent execution of simultaneously occuring events in the simulated world enhances the quality of disaster-rescue simulation giving realistic results. Lastly, the speed-up provided by porting the simulator code on GPU is at least ten thousand times against a CPU of comparable configuration. This allows us to model one second of the real world clock as one cycle of the disaster-rescue simulation, thus giving a more detailed and fine-grained simulation with even lesser amount of time taken than the current simulator.

1.1 Background and Related Work

The RoboCup Rescue Simulator [4] consists of five modules : *Kernel, agents, component simulators, GIS* (geographical information system) and *viewers*. Individuals in the simulation system are modelled as agents, number of agents in the current simulator being limited to a few thousands. *Kernel* is responsible for managing all the communication between agent modules (agents cannot communicate directly; but can communicate via *Kernel*). *Kernel* receives commands from different agents, filters them, and broadcasts them to *component simulators*. *Component simulators* correspond to various simulation domains, such as earthquakes, fires, logistics or traffic jams and compute what will happen in the world based on current events and activities. These computations are sent to the *Kernel*, which integrates the results from several *component simulators* and broadcasts the result to the simulators and GIS. *Kernel* waits for a fixed period of time, $\in$(as it is a real-time simulation), to receive computations from the simulators. If some of the simulator messages fail to reach within this time limit, the *Kernel* may present a wrong picture of the world to these simulators and GIS.

A lot of work has been done in improving the code organization and in development of RoboCup Rescue Simulator. Several improvements have been made since the first competition held in RoboCup, 2001 - automatic validation of maps on *Kernel* startup, ability to run the simulation step-by-step or automatically, new tools for generating maps and scenarios have been developed and design of communication protocols are made more robust. The current version [6] of the simulator has these improvements.

However, the dependance on main memory is still a problem with the current simulator and limits the number of agents in the simulation. This problem has been addressed in [3], in which a database driven model of the rescue simulator is provided. This model eliminates the dependancy on main memory by modelling each entity in the simulation as a relational table. But as the number of agents are increased to a few hundred thousands, the model incurs significant latency costs in accessing the database resident in the hard disk. Moreover, it still uses a cycle-based approach for the execution of agents, which is against the true nature of a multi-agent simulation.

1.2 Motivation for Using GPUs

GPUs are based on multiprocessors each with 8-10 cores and hundreds of ALUs, several thousand registers and some shared memory. Besides, a graphics card contains fast global memory (which can be accessed by all multiprocessors), local memory in each multiprocessor, and special memory for constants. Different from multicore CPUs, the cores on the GPU are virtualized, and GPU threads are managed by

the hardware. Such a design simplifies GPU programs and improves program scalability and portability, since programs are oblivious about physical cores and rely on hardware for thread creation and management. Most importantly, these several multiprocessor cores in a GPU are SIMT (single instruction, multiple thread) cores and execute the same instructions simultaneously. This programming style is useful for multi-agent simulations in which we have same code for same type of agents. Further, GPU threads are lightweight threads and do not require much context switching. Whenever a request for a thread comes in, an idle thread is picked up from the formerly created pool of threads. Switching from one thread to another costs hundreds of cycles on CPUs, but GPUs switch several threads per cycle.

Multiple GPU devices can be ported on a single processor. The different *component simulators* are run on separate devices and *Kernel* on a separate device. Since all devices are ported to the same CPU, the transfer of percept information need not be serialized and no string manipulations need to be done. These factors further enhance the performance and take up the number of simulated agents to a billion (with thirty GPU devices plugged-in).

The GPUs, available today, are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as 'C'. nVidia Compute Unified Device Architecture (CUDA) [1] is a technology that enables programmers and developers to write software to solve complex computational problems in a fraction of the time by utilizing the many-core parallel processing power of GPUs. It provides a 'C'-like programming language, 'C for CUDA' (C with nVidia extensions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU.

2 nVidia Compute Unified Device Architecture

The CUDA architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions. Threads are organized in two- or three- dimensional thread blocks. Threads within a block can co-operate among themselves by sharing data through a shared memory and synchronizing their execution to co-ordinate memory accesses. These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processors in the system. Number of thread blocks can greatly exceed the number of processors.

A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multi-threaded instruction unit, and on-chip shared memory. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. It implements the barrier synchronization intrinsic with a single instruction. Fast barrier synchronization, together with lightweight thread creation and zero-overhead thread scheduling, efficiently support very fine-grained parallelism, allowing, for example, a low granularity decomposition of problems by assigning one thread to each data element (in present case, assigning a thread to each agent). To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture, SIMT (single-

instruction, multiple-thread). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.

CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory. Each thread block has a shared memory (16kB in size) visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages.

3 nVidia CUDA's Programming Model

'C for CUDA' extends C by allowing the programmer to define C functions, called kernels [1], that, when called, are executed N times in parallel by N different CUDA threads.

A kernel is defined using the *__global__* declaration specifier and the number of CUDA threads for each call is specified using a new <<< ... >>> syntax. The dimension of the grid is specified by the first parameter of the <<< ... >>> syntax, while the second parameter specifies the number of threads to be invoked.

Each of the threads that execute a kernel is given a thread identifier that is accessible within the kernel through the built-in *threadIdx* variable. Similarly, each block in the grid is given a unique identifier accessible through variable *blockIdx*. *threadIdx* and *blockIdx* both start from index 0 and together they uniquely identify a thread by the relation (*blockIdx * blockSize + threadIdx*). A kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or serially. This independence requirement allows CUDA to schedule thread blocks in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

CUDA's programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels (also referred as *device code*) execute on a GPU and the rest of the C program (referred as *host code*) executes on a CPU. Further, it assumes a system composed of a *host* (CPU) and a *device* (GPU), each with their own separate memory. Kernels can only operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory. In order to facilitate concurrent execution between *host* and *device*, kernel launches are made asynchronous: control is returned to the host thread before the device has completed the requested task.

The number of blocks a multiprocessor can process at once, referred to as the number of active blocks per multiprocessor, depends on how many registers per thread and how much shared memory per block are required for a given kernel since the multiprocessor's registers and shared memory are split among all the threads of the active blocks. If there are not enough registers or shared memory available

[1] To avoid confusion between CUDA kernel and the *Kernel* of simulator, the latter is always capitalized and italicized.

per multiprocessor to process at least one block, the kernel will fail to launch. The number of registers required per thread (for a given kernel code), shared memory required per block and its effect on performance can be determined using CUDA Occupancy Calculator [2] provided by nVidia CUDA.

A host system can have multiple devices. These devices can be enumerated, their properties can be queried, and one of them can be selected for kernel executions. Several host processes can execute device code on the same device, but by design, a host process can execute device code on only one device at any given time. As a consequence, multiple host processes are required to execute device code on multiple devices. Also, any CUDA resource created through the runtime in one host process cannot be used by the runtime from another host process.

4 Programming RoboCup Rescue Simulator with CUDA

4.1 Modelling Agents

We model each agent, both civilian and rescue agents, as a separate thread on the GPU, with agent identifier being uniquely assigned as *(blockIdx * blockSize + threadIdx)*. Agent code is written as the kernel function and GPU executes this code in parallel threads, one thread for each agent. Therefore, agents of the same type share the same kernel code.

4.2 Programming Simulator *Kernel*

The *Kernel* runs several number of threads, known as *computation threads*, for processing agent modules' messages and for computing the sensory information for each agent in each cycle of the simulation. If there are N agents in the simulation and *Kernel* is running M *computation threads*, then each *computation thread* is responsible for processing requests for $\lceil N/M \rceil$ agents. Each agent is mapped to a *computational thread* and the same *computational thread* processes the requests for this agent in every simulation cycle. In an ideal scenario, M is kept equal to N. The latency of computing perceptions for each agent serially is a major bottleneck in the current version of RoboCup Rescue Simulator. The proposed simulator on GPU does away with this latency through the parallel *computational threads*. The perception computation which initially had a complexity $O(N)$ has the new complexity as $O(\lceil N/M \rceil)$. The code for computing the perceptions of different agents is specified as the kernel code for execution by the *computational threads* of the simulator *Kernel*.

4.3 Programming *Component Simulators*

The simulation system consists of D number of GPU devices if there are *(D-1)* simulation domains, with each domain being run by a single *component simulator*. *Kernel* and *component simulators* are run on separate devices. The device which runs *Kernel* has no agent threads running on it and executes only the *computational threads*. The devices running *component simulators* has the agent threads being divided among them equally.

The computations done by a *component simulator* can also be easily parallelized. The civilian simulator creates different agent threads for each civilian, each running the civilian agent code in the kernel function. The fire simulator computes the state of the environment by dividing the city into equal areas and does the

computation for each of the areas in parallel threads. The number of divisions depend on size of the city and complexity of the *component simulator's* simulation domain.

4.4 Execution of the Simulation Cycle

Steps in the simulation cycle are basically the same as in the current version of the RoboCup Rescue Simulator; the changes are in the way we execute them on the CUDA driven GPUs. Cycle begins with the *Kernel* computing the sensory information for each individual agent in separate concurrent *computational threads*. The information computed is represented as an appropriate data structure containing data elements for each of the parameters in accordance with the protocols being followed in the current version of the simulator, and passed to the thread running the concerned agent's code. Since, all the messages are being passed between devices running on the same host CPU, there is no need for serializing [2] the parameters for sending messages from one simulator module to another. This removes the latency in serializing and deserializing of agent data, which takes considerable amount of execution time in the current version of simulator.

Next, the agent threads receive the sensory information from the corresponding *computational threads* of the *Kernel* and decide what actions to take and send the action commands to the *Kernel*. The *Kernel* gathers messages from all the agent module threads and broadcasts them to the *component simulators*. The *Kernel* handles these messages in the corresponding *computational threads*.

The *component simulators* running on separate devices compute how the world changes with the updated information received from the *Kernel* and returns the result back to the *Kernel*. The *Kernel* integrates the results received from the different *component simulators*, and broadcasts them to the GIS and the simulators. The *viewers* request the GIS to send updated information of the world and display visually the information. CUDA supports OpenGL and its novel architecture does a fast rendering on the graphics window.

4.5 Some useful tools provided by CUDA runtime

Passing the information between *Kernel* and *component simulators* require data transfer from one device to another. CUDA provides *cudamemcpy()* for transferring data between two devices or between a host and a device. With the Dual PCI-Express, the bandwidth between the host processor and the Tesla processors is maximized up to 12.8 GBytes/sec transfer rate and device-to-device upto 100 GBytes/sec transfer rate.

Synchronization of threads at the end of each cycle is done by specifying synchronization points in the kernel by calling the *__syncthreads()* intrinsic function; *__syncthreads()* acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

Since a number of threads are running in parallel, scenarios may arise when different threads try to change the values stored at the same memory location (same environment variables) simultaneously, giving rise to race conditions. This is a common case when different threads of the different component simulators attempt to modify the state of the same object in the city. In such cases there

[2] Currently serialization needs to be done for transferring information between two simulator modules running on different CPUs.

is a need to do atomic writes and have proper synchronization between threads. This can be handled using *atomic* functions provided by the CUDA. An *atomic* function performs a *read-modify-write* atomic operation on one 32-bit or 64-bit word residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads.

The runtime also closely monitors the device's progress and performs accurate timing, by letting the application asynchronously record events at any point in the program and query when these events are actually recorded. An *event* is recorded when all tasks, or optionally all commands in a given stream, preceding the *event* have completed. This helps in simulating real-time problems which require response from various entities within a fixed amount of time. Further, it provides a mechanism to maintain the simulation clock which is useful for cases when we need to log the progress of the simulation for handling failures. By copying the data from the device memory to host memory and using this log information, we can restart the simulation from the point of failure.

5 Results

For validating the utility of porting the RoboCup Rescue Simulator code on GPUs, we coded three dummy simulators - traffic, earthquake, civilian - and a dummy *Kernel*. Each agent was assigned a data-structure of size 250 bytes for defining its state. Execution follows the cycle as described in the previous section. Every agent sends random messages to 1000 other agents in every simulation cycle and no agents ever die; this is done so as to ensure maximum stress on the simulator at all time. The *component simulators* receive *GIS* and agents information from the *Kernel*. These simulators do not compute anything but sleep for 0.25 seconds before responding to the *Kernel* so as to give a feel that some computation is being done by them. The current version of Rescue Simulator executes 500 agents in 1 second per simulation cycle when running on 4 CPUs (1.67 GHz and 2 GB RAM). In accordance to the current simulation cycle as described in [4], we can reasonably estimate the time taken by a *component simulator* to compute the next state of the world to be not more than 0.25 seconds. Hence, the sleep time is justified. In the dummy simulator, agents receive the same perceptual information every time and hence always give a command to the *Kernel* to move by zero unit. As there is no change in the state of the world, *Kernel* computations bring out the same percepts every time. We ran the simulation with *9 million* civilians and *1 million* rescue agents.

For experiments we used 5 nVidia Tesla T10 GPUs each with 933 GFLOPS of processing performance, 1.30 GHz clock-rate and 4 GB of GDDR3 memory at 102 GB/s bandwidth. It has 30 multi-processors with 240 cores, a constant memory of 64 MB, a shared memory of 16kB per block and 16k registers per block.

Each *component simulator* was assigned one GPU device, one device was assigned to the *Kernel* and the one left was used only for running the agent software. The GPU devices running *component simulators* were allotted 2 million civilian agent threads each to run on them (in addition to computing the corresponding simulation domain). Rest of the agents were allotted to the device dedicated for running the agent software. The number of *computational threads* on the device running *Kernel* was set to 5 million. The block size was kept as 512 threads. Grid size for a device was computed as $\lceil (K/512) \rceil$, where K is the number of agents allotted to that device.

It took *453 milli-seconds* for running one simulation cycle with *10 million agents*, on an average computed over 100 simulation cycles. Scalability provided is approximately 10^4 *times* ($10^7/10^3$) times over the current version of RoboCup Rescue Simulator. If the current version of simulator was scalable enough to execute 10^7 agents, it would have taken $1 \times 10^7/500 = 2 \times 10^4$ seconds to complete one simulation cycle; approximately $0.80 \times 2 \times 10^4/0.453 = 3.5 \times 10^4$ times speedup is achieved (0.80 factor comes in because 4 CPUs are compared with 5 GPUs). Further, a low simulation cycle time of 453 milli-seconds encourages us to model 1 second of real-world clock as one simulation cycle.

5.1 Discussion

GPUs are designed to provide tremenduous concurrency. High performance and fast memory bandwidth together with light-weight context switching between threads make it a promising tool for parallelization of modular problems.

Parallelism and modularity is inherent in the conceptual architecture of RoboCup Rescue Simulator. Computation of percepts for each agent is independent of others. The events occuring and its impact on a particular area of the city can be computed for individual areas separately. Finally, an agent itself is an autonomous entity with its own intelligence and abilities to take actions. Further, since all GPU devices running the simulation are ported on the same host CPU, serializing and deserializing of objects used for sending percepts information is not required. Further, messaging between GPUs (Dual PCI Express with 102 GB/s) hosted on the same processor is faster as compared to between two different CPUs over a Gigabit ethernet. These reasons, together with the results obtained, encourage us to parallelize the RoboCup Rescue Simulator code on fast, scalable GPUs.

6 Conclusion

In this paper, we presented the possibilities of performance and scalability enhancements that can be achieved on porting the current RoboCup Rescue Simulator architecture on CUDA driven nVidia GPUs. Apart from speeding up the simulation by more than ten thousand times against CPUs, it facilitates a more detailed and finer-grained simulation with each simulation cycle representing 1 second of the real-world clock. Another major improvement is that the number of simulated agents is scaled up to ten million.

Using CUDA has its limitations. CUDA only supports nVidia graphics card and is not compatible with others (for example ATI graphics cards). Therefore, our future aim is to develop a framework which can run on a graphics card of any make and thus build up a distributed system of heterogeneous GPUs. We aim to utilize this distributed system to develop the RoboCup Rescue Simulator.

References

1. nVidia CUDA home-page : [http://www.nvidia.com/object/cuda_home.html]
2. CUDA Occupancy Calculator : [http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls]
3. Rahul Sarika, Harith Siddhartha, and Kamalakar Karlapalem - Database Driven RoboCup Rescue Server - RoboCup, 2008
4. Tomoichi Takahashi, Ikuo Takeuchi, Tetsuhiko Koto, Satoshi Tadokoro, Itsuki Noda : RoboCup-rescue disaster simulator architecture : RoboCup 2000: Robot Soccer World Cup IV
5. RoboCup Rescue Simulator Documentation [http://www.robocuprescue.org/documentation.html]
6. RoboCup Rescue Simulator Version 1 [http://sourceforge.net/projects/roborescue/]