ISBN: 978-989-20-2190-4

Mapping Multi-Agent Systems Based on FIPA Specification to GPU Architectures

Luiz Guilherme Oliveira dos Santos[1], Flávia Cristina Bernadini[1], Esteban Gonzales Clua[2], Luís C. da Costa[2] and Erick Passos[2]

[1]Departamento de Ciência e Tecnologia (RCT)
Polo Universitário de Rio das Ostras (PURO)
Universidade Federal Fluminense (UFF)
Rio das Ostras, RJ – Brazil

[2]Instituto de Computação (IC)
Universidade Federal Fluminense (UFF)
Niterói, RJ – Brazil

santos.uff@gmail.com, fcbernardini@vm.uff.br, esteban@ic.uff.br,
lcosta@ic.uff.br, erickpassos@gmail.com

Abstract

The dynamic nature of agents distribution motivates the investigation on the standardization of dynamic collaborative multi-agent systems (MAS). Among many different models available to implement a MAS, one of the most commonly used by developers community is FIPA. We've got many frameworks that adheres FIPA specification, JADE for instance. A MAS has the interesting property to allow the sub-problems related to a constraint satisfaction problem to be subcontracted to different problem solving agents with their own interests and goals. In some situations, it is necessary to simulate a crowd of agents using MAS. However, crowd agents simulation with real time performance are difficult to be achieved without a huge parallel architecture. On the other hand, in the recent years, parallel architectures are becoming massively available. One example of this kind of architecture is CUDA, a GPU architecture that allows to parallelize general algorithms and run them at a GPU level. In this work, we investigate and propose how to map MAS implementation from JADE framework to CUDA. We also develop and implement a novel process, in which we could verify a crowd simulation that was not possible to be executed using JADE framework. The obtained results in our simulations were very promising, but we still looking foward for new case tests to specify the best situations in which using a GPU archtecture is the best way to implement a MAS.

Keywords: FIPA, GPGPU, MAS, Artificial Intelligence

1. Introduction

Multi-Agent Systems (MAS) comprehend an approach to build complex distributed applications. A MAS consists of a population of autonomous entities (agents) situated in a shared structured entity (the environment) (Franklin, 1996). Indeed, in literature we can observe that researchers agree that agents are entities within an environment, and that they can sense and act (not necessarily in that order). The dynamic nature of agent distribution motivates research by groups working on the standardization of dynamic collaborative multi-agent systems. Mendez (Flores-Menderez, 1999) describes each model proposed by these groups, and concludes that "The architecture models open environments composed of logically distributed areas where agents exist. The basic agents in this architecture are minimal agents, local area coordinators, yellow page servers, and cooperation domain servers". One of these models, used in this work, is Foundation for Intelligent Physical Agents (FIPA).

A MAS has the interesting property to allow the sub problems of a constraint satisfaction problem to be subcontracted to different problem solving agents with their own interests and goals. Furthermore, domains with multiple agents of any type, including autonomous vehicles (Stone 1997) and even some human agents, are studied. We're trying to create a full description of FIPA-GPGPU Mapping. Related works like (Richmond, 2009) explore this mapping as well, but specify the agents using XML language. As a result of our work we would like to give the programmer a set of tools to create agents in GPU using FIPA specifications.

The purpose of this work is to explore how to map MAS from a distributed to parallel software and hardware architecture. We describe how a MAS is usually implemented following the FIPA model, and what is concerned to implement a MAS in GPGPU based architectures. We also describe a case study to explore how to map a MAS implementation following FIPA model to GPGPU language, using CUDA in our case.

This paper is organized as follows: Section 2 describes Multi-Agent Systems in general, FIPA as well, and give an overall about frameworks that follows its specification. Section 3 describes the GPGPU Paradigm. Section 4 describes a case study we developed to evaluate and analyze the mapping problem between a FIPA based implementation and a GPGPU based implementation. Section 5 describes how the mapping process occurs to our case study, and the results obtained when measuring time execution of the implementations. Finally, Section 6 concludes this work and describes future work.

2. Multi-Agent Systems and FIPA

Software agents are computational actors that act performing a defined script. Another definition to agent is everything that is capable to observe its environment using a sensory input and to act in this environment by effectors (Russell, 1995). When many agents interact among each other in an environment, the system is called Multi-Agent System. Nowadays, its use is as variable as possible, from “robots” that search pieces of information in websites, to evacuation and massive people simulations (Valckenaers, 2006).

There are many frameworks that simplify the implementation and modeling of interoperable intelligent MAS. These frameworks simplifies the development of the agent-based applications, providing a middleware system implementing the life-cycle support features required by the agents, communication directives, behaviors methods and some graphical tools to administrate the system. In this work we choose to work with JADE (Java Agent DEvelopment Framework)(Bordini, 2006; Bellefimine, 2008), because it has huge set of libraries and tools that are provided to this language, its paradigm is changed to an Agent-Oriented programming (Bellefimine, 2007), that brings the concepts and definitions of years of research into the Artificial Intelligence area.

3. The GPGPU Paradigm

Differently from a superscalar CPU, a GPU (Graphic Processing Unit) is made specifically for massively parallel floating point computations, the king of operations that are the bottlenecks on computer graphics, which was the original purpose of this type of hardware. Because of this specialized nature, a GPU reserves little space for control and cache, but reserve a lot of resources to Arithmetic Logic Units. However, with the development of programmable graphic units some years ago, it also became possible the use of GPU’s for general purpose computing. Being massively parallel devices, large sets of parallel computational problems are suitable to being solved in these architectures, giving birth to the concept of GPGPU (General Purpose Graphic Processing Unit).

In this paradigm, the input data is processed as a stream, i.e, all the information will be processed in parallel chunks as its stream is fed into the GPU’s processing units, each one executing a copy of the same code. We call this kind of code a Kernel Function, which operates in a data-parallel fashion over the data stream. Besides designing the algorithms and data-structures to stream computing, one of the main difficulty with the GPGPU paradigm in the past was that one had to map them as Shaders and Pixels respectively, which sometimes is impracticable and demands a very difficult learning curve. In 2006, NVIDIA unified the architecture of its GPU's and gave programmers access to it through extensions of the C and FORTRAN languages. This architecture is called CUDA — Computer Unified Device Architecture (NVIDIA, 2007), and now several similar proposals followed the same idea (Group, 2009; Ghuloum, 2007).

It should be observed that GPGPU has many restrictions that impose obstacles for the direct mapping from a traditional procedural or Object-Oriented paradigm. Its architecture provides the concept of threads, which are composed into blocks, and these blocks are organized into a grid. The number of threads and blocks that can be created varies according to the hardware that is used in the problem and it is not possible to use recursion. Even nested loops should be avoided. All algorithms must be designed to use as many threads and less loops or branches as possible. To make it suitable and practical to stream processing, it is mandatory to design the algorithms as data-parallel kernel functions. The general approach makes use of the indexes of the threads and blocks to access data locally and to distribute the computing among them. When a CUDA kernel is invoked, all threads execute the same code, but they'll normally process a different chunk of data.

4. Case Study – Box Problem

The problem is concerned to move boxes from one string to another. Each box moved from a position of the first string should be moved to the second string in the same position of the former. The number of boxes is bigger than the number of agents. Agents are put in random positions between the two strings, and they have 3 (three) possible actions: they can move to the left and to the right and they can move a box when there is a box in the same position of the first string. Each agent can move the box from the first to the second string, as can be seen in Figure 1, and after, it can move to the left or to the right. This is considered a simple

Algorithm 1 Algorithm to move the boxes.

```
Require: Pos: Position of the box in one line.
1: while All the boxes have not been moved do
2:    if There is a box on the right side then
3:       MoveTheAgent(right);
4:    else if There is a box on the left side then
5:       MoveTheAgent(left);
6:    else
7:       MoveTheAgent(random);
8:    end if
9: end while
```

Figure 1: Algorithm to Move the Boxes

problem, because we have just one type of agent, our agents have only three possible actions and they do not need to communicate with each other.

5. Mapping from FIPA to GPGPU

Since JADE follows FIPA specifications and map each agent as a single thread in a container of agents, an implementation of a MAS obeying these specifications should be implemented in JADE. One of the main problems we could detect to implement a MAS in CUDA is the different paradigm adopted by each one. In JADE, all the agents are objects with behaviors programmed as a method of the Agent's Object. CUDA, however, needs lots of data structures and functions to map the agents and its behaviors. All agents were mapped into a vector and, when the kernel is launched, each thread of the kernel controls one agent by using a pre-defined function. In problems where different types of agents with different behaviors are available, we need to identify the agent using more parameters.

5.1 Implementation in JADE

As JADE uses the Agent-Oriented paradigm, the agents were mapped as an object that has a private class called **PersonBehavior**. This class is responsible to determine the agent's actions and verify if the agent's work is done. To do it there are 2 methods called **action()** and **done()** respectively. On the top of it, there's a main class that creates the scenario and put randomly the boxes on the spots. This main class creates several agents as well, in an Agent Container provided by the JADE's framework, and put them in random spots like the boxes. The JADE run-time creates a Main Container to execute this container of agents. Following the Algorithm 1, all the agents moves the boxes and verify if all the boxes have been moved on the method **done()**, if it is, their work is finished and the agents are killed.

5.2 Implementation in CUDA

Although CUDA is easy to get started and improves the performance of the GPU for general purposes, its restrictions made the mapping from JADE not so easy. One of the most difficulties we had was changing from the Agent-Oriented paradigm to the GPGPU paradigm. All the agents were mapped as a vector, the index of this vector indicates the agent's ID and its value indicates the agent's position. A *global function* [1]called kernel creates several threads

[1] In CUDA, *global functions* are the ones who access the GPU directly from the CPU

for each agent, each thread indexing a position of the vector. Before starting the execution, we have to allocate all the initial information on the GPU's memory.

Each thread runs the algorithm shown in Figure 1. Notice that the kernel is our **action()** method from JADE's implementation. If we have different kind of agents in this scenario, we need to map this problem differently. We also have a similar function to the method **done()**, that analyses when all the boxes are moved. This function is a *device function*[2], and all the threads has access to it. When all the packages are moved, the kernel finishes, and new data should be allocated from the GPU to the CPU.

The biggest advantage is that CUDA created all the threads at the same time, differently from JADE. In JADE, it was created the Agent Container, and then all threads were created in this container. In CUDA, all the variables that were created on the CPU, such as the agent's vector and the scenario, must be allocated on the GPU's memory, that demands some time. Another CUDA's Disadvantage is that if there is too much communication between the agents, we need to sync the kernel execution many times, compromising the scalability performance.

5.3 Performance and Analysis

In this work we evaluated two multithread implementations, one using JADE and another using CUDA. The tests were performed on a Intel Core 2 Duo 2.26GHz CPU, 4GB of RAM and in a NVIDIA Geforce 9400M GPU with 256MB of memory.

Scn	# Agents	# Boxes
1	250	500
2	500	500
3	250	1,000
4	500	1,000
5	1,000	1,000
6	250	2,000
7	500	2,000
8	1,000	2,000
9	1,500	2,000
10	2,000	2,000
11	250	5,000
12	500	5,000
13	1,000	5,000
14	1,500	5,000
15	2,000	5,000
16	5,000	5,000
17	250	1,0000
18	500	1,0000
19	1,000	10,000
20	1,500	10,000
21	2,000	10,000
22	5,000	10,000

Table 1: Case Test Specifications

We tested 22 different instances of the problem, which we call as scenario, and each experiment scenario was executed 10 (ten) times. Table 1 shows each experiment configuration, where Scn means a scenario identification, # Agents means the number of

[2] In CUDA, *device functions* are only accessed on the GPU by functions that are running on it.

agents used in the experiment, and # Boxes means the number of boxes used to be moved by the agents. We calculated the speed-up between JADE implementation and CUDA implementation for each experiment. We call speed-up the time taken by the JADE implementation to move all the boxes by the agents divided by the time taken by the CUDA implementation to execute the same task. We calculated the mean and standard deviation of

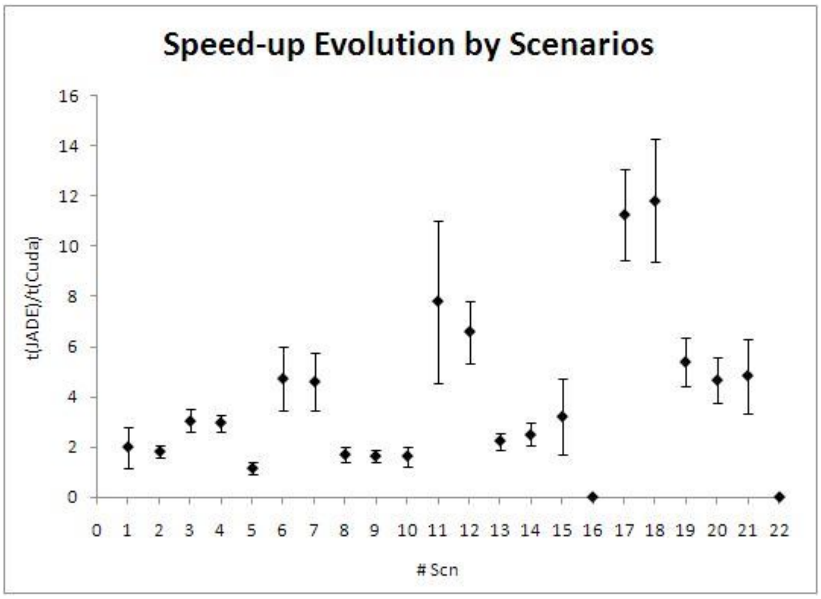

Figure 2: Speed-Up graphic. X-axes refer to each experiment scenario and Y-axes refer to mean of speed-up in three executions. Vertical lines in each point refer to standard deviation of the mean.

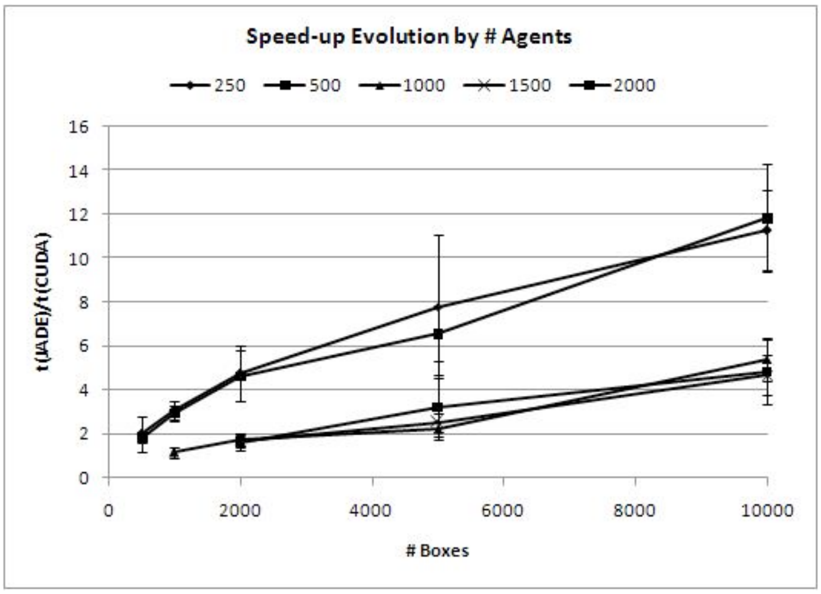

Figure 3: Speed-Up evolution by number of agents.

the measured speed-up of the 10 (ten) execution times obtained for each experiment.

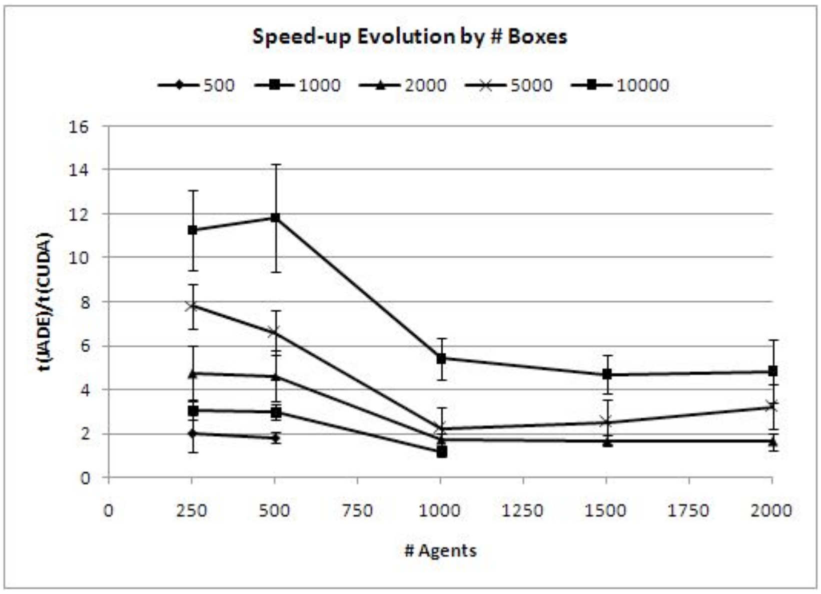

Figure 4: Speed-Up evolution by number of boxes.

Figure 2 shows a plot in which each point is the mean and standard deviation values of the speed-up for each experiment. In this figure, we can observe that the speed-up in scenarios 16 and 22 is 0. This is because our JADE implementation could not create more than 5,000 agents, and we do not considered the related times for these cases. The scenarios with 5,000 agents or more could not be correctly executed since there was not available memory[3]. On the other hand, in case 17, the CUDA implementation is 11 times more efficient than the JADE implementation. Also, using CUDA, in scenarios 21 and 22, the time execution were respectively 297 ms and 425 ms, turning possible crowd simulations using MAS implementations. These results show that implementing SMAs using FIPA specifications in CUDA is very encouraging when the agents' work is parallelized.

In order to improve our analysis, since scenarios 16 and 22 could not be executed in our JADE implementation, we unconsidered 5000 agents and we plotted graphics to analyze how speed-up evolves when separating both variables — number of agents and number of boxes. Figures 3 and 4 respectively show these graphics. Figure 3 shows that when increasing the number of agents, speed-up grows; and Figure 4 shows that when increasing the number of boxes, speed-up decays until a certain point, and then become stable (with 1000 boxes). The speed-up evolution shown in Figure 3 were expected, but the one shown in Figure 3 were not, although all speed-ups were greater than 1 (one), indicating that CUDA was faster than JADE

[3] The error obtained in our executions with 5,000 agents were "Unable to create new native thread.". However, simulations with 2,000 agents were sufficient to evaluate our mapping.

in all scenarios, as observed before in Figure 2. Analyzing why the speed-up decays, we observed that this is because CUDA is faster than JADE when it creates its threads. In JADE we had to manually create each thread using a loop, in addition to the threads necessary to control the execution. However in CUDA, when we call the kernel function, it creates all the necessary threads to execute de code. We expect that using a more powerful GPU, this stable point will be further than on these tests.

These results were expected, since the problem we explored was used to evaluate CUDA performance. We're looking forward for more computational complex MAS problems, like pathfiding and evacuation simulation problems, in order to improve our method to map FIPA-GPGPU problems.

6. Conclusions and Future Work

In this work, we show two different paradigms — Agent-Oriented and GPGPU — to develop a MAS. We compare their differences, advantages and disadvantages. We also present how to map the same algorithm from the Agent-Oriented paradigm using JADE's framework to a GPGPU paradigm presenting a case study, and the difference of performance from one implementation to the other one. We could observe that when we a simulation with many agents using JADE, the simulations could not be executed. Since we want to turn possible to simulate in real time complex crowd simulations, we think that using GPGPU paradigm, such as CUDA, is a good way to solve this problem.

Since a simple problem could not be simulated with so many agents in JADE due to limits of required memory in our tests, we will implement our scenario in CUDA. In order to improve our mapping from JADE to CUDA, we intend to implement this scenario also in JADE to maintain our methodology. We believe that CUDA is more useful when we have many similar agents with less communication, although JADE is better to map when we have many different types of agents and it is not necessary to simulate a huge number of agents. However, we need to evaluate our beliefs, and this is the next step of our work.

References

Bellifemine F., Caire G., Greenwood D. (2007): Developing multi-agent systems with JADE. Wiley Series in Agent Technology

Bellifemine F., Caire, G., Poggi A., Rimassa G. (2008): Jade: A software framework for developing multi-agent applications. lessons learned. Infomation and Software Technology, 50, 10–21

Bordini R., Braubach L., Dastani M., Seghrouchni, A.E.F., Gomez-Sanz J., Leite, J., OHare G., Pokahr A., Ricci A. (2006): A survey of programming languages and platforms for multi-agent systems. Informatica, 30, 33–44

Dimakis N., Filippoupolitis A., Gelenbe E. (2009): Distributed building evacuation simulator for smart emergency management. The Computer Journal, Available at http://sa.ee.ic.ac.uk/publications/DBES_CJ.pdf

Flores-Menderz R. (1999),: Towards a standardization of multi-agent system frameworks. ACM Crossroads Magazine http://www.acm.org/crossroads/xrds5-4/ multiagent.html

Franklin S., Graesser A. (1996): Is it an agent or just a program? A taxonomy for autonomous agents. In Muller, J., Wooldridge, M., Jennings, N., eds.: Intelligent Agents III. Agent Theories, Architectures, and Languages. LNAI. Volume 1193, 21–35

Ghuloum A., Smith T., Wu G., Zhou X., Fang, J., Guo, P., So B., Rajagopalan M., Chen Y., Chen B. (2007): Future-proof data parallel algorithms and software on intel multi-core architecture. Intel Technology Journal 11(4)

Group K.(2009): OpenCL overview. http://www.khronos.org/developers/library/overview/opencl_overview.pdf

NVIDIA(2007): CUDA description. http://www.nvidia.com/cuda

Richmond P.; Coakley S., Romano D. M.(2009): A high performance agent based modelling framework on graphics card hardware with CUDA. In: Proc. 8th International Conference on Autonomous Agents and Multiagent Systems. Vol. 2, pp 1125-1126

Russel S., Norvig P. (1995): Artificial Intelligence, A Modern Approach. Prentice Hall, Second Edition.

Stone P., Veloso M. (1997): Multiagent systems: A survey from a machine learning perspective. Autonomous Robotics, 8, 345–383

Valckenaers, P., Sauter, J., Sierra C., Rodriguez-Aguilar J. (2006): Applications and environments for Multi-Agent Systems. Autonomous Agents and Multi-Agent Systems 14(1), 61–85