Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors

Roman Pavlov, Jörg P. Müller

Multi-agent Systems meet GPU: Deploying Agent-based Architectures on Graphics Processors

Roman Pavlov[1], Jörg P. Müller[1]

[1] Clausthal University of Technology, Clausthal-Zellerfeld, Germany
{roman.pavlov, joerg.mueller}@tu-clausthal.de

Abstract. Even given today's rich hardware platforms, computation-intensive algorithms and applications, such as large-scale simulations, are still challenging to run with acceptable response times. One way to increase the performance of these algorithms and applications is by using the computing power of Graphics Processing Units (GPU). However, effectively mapping distributed software models to GPU is a non-trivial endeavor. In this paper, we investigate ways of improving execution performance of multi-agent systems (MAS) models by means of relevant task allocation mechanisms, which are suitable for GPU execution. Several task allocation architecture variants for MAS using GPU are identified and their properties analyzed. In particular, we study three cases: Agents and their runtime environment can be (i) completely on the host (CPU); (ii) partly on host and device (GPU); (iii) completely on the device. For each of these architecture variants, we propose task allocation models that take GPU restrictions into account.

Keywords: Multi-Agent Systems, GPGPU, CUDA.

1 Introduction

Today's software-intensive networked applications, such as e.g. microscopic traffic simulations 1 cause increasing demand for computational resources. Moreover, Ubiquitous Artificial Intelligence (UAI, 5) applications such as decentralized energy systems control 2 or cooperative traffic management 3, are data-intensive: the amount of data steadily increases, and more powerful algorithms and more computational units are needed, while paying attention to better performance per Watt.

A state-of-the-art approach to this challenge is to make use of recent progress in parallel and distributed computing. Distributed computing helps divide tasks across different computational units. If well coordinated, it can lead to improved performance and scalability performance improvement, as it provides a possibility to decrease computational time by using multiple machines. However, it trades off computation with communication overhead, thus Amdahl's Law 4 and the specifics of the application at hand need to be taken into account.

Multi-Agent Systems (MAS) 8 are a paradigm, which is often used for UAI modeling and exploits both distributed and parallel computing. A MAS represents a system as a set of interacting agents - software units that are able to execute tasks for reaching local (or system) goals. Agents co-exist in the environment, and they communicate with each other to exchange information and coordinate task execution. MAS can be applied to find solutions for complex problems by decoupling them into less complex subproblems and providing coordinated solution frameworks such as the contract net protocol 25 or simulated trading 28. MAS are of particular interest when viewed as a modeling and simulation paradigm 27. The MAS metaphor enables fine-grained modeling of systems with local preferences, motivations, and capabilities, including issues of interaction, coordination, and cooperation. This is of use in many large-scale networked simulation tasks, such as traffic simulation 6 or physical, ambient intelligence simulations. However, a big challenge for agent-based simulation is that they are highly computation-intensive, which to date limits their scalability compared to e.g. approaches based on system dynamics.

Thus, a key objective of this work is to make a contribution to more scalable, high-performing agent-based simulation. In this paper, we focus on a specific aspect of MAS technology, i.e. on the question of how tasks in a MAS are assigned to agents.

Typically, task allocation and scheduling in MAS is achieved by dedicated protocols for cooperative problem solving, such as the contract net protocol 25, which defines how problems are decomposed, assigned to agents, and scheduled for execution. Our hypothesis is that we can exploit information related to task decomposition, allocation, and scheduling contained in MAS models for efficiently mapping these models to models (code) executable on a parallel (e.g., graphics) hardware.

Modern computers are highly parallel. Their Central Processing Units (CPU) have more than one core. In addition, most of the modern computers have Graphics Processing Units (GPU) that can be used as a coprocessor for computational tasks. This becomes possible because of frameworks such as CUDA/OpenCL, which support General-Purpose Computing on Graphics Processing Units (GPGPU) 7.

This paper introduces an approach to tasks scheduling in MAS; our approach couples MAS technology with GPGPU programming. We present three types of task scheduling models for MAS that uses GPU for non-graphical tasks. Our goal is to establish a correspondence between agents (which plan and execute computational tasks) and massively parallel, GPGPU-capable computational resources.

The structure of this paper is as follows: Section 2 describes the relationship to Internet of Things; Section 3 outlines the current state-of-the-art; Section 4 shows the research contribution and innovation of this work. In Section 5 we critically discuss the results; Section 6 gives conclusions and directions for future work.

2 Relationship to Internet of Things

Internet technology is becoming ubiquitous. This concerns the service layer as well as the network layer. At the same time, virtualization is used to provide network users with a homogeneous interface to ubiquitous, decentralized services. Cloud computing

is one example of this trend. MAS is a suitable metaphor to describe decentralized environments and applications where large numbers of semi-autonomous, self-interested or cooperative entities (services, objects, devices) interact and solve problems by collaboration and cooperation. Agents are considered as Smart Objects 24 that have their own behavior. MAS have been successfully applied to a range of problems, e.g., simulations (see e.g. 9). When it comes to simulation, MAS is a very interesting paradigm for the implementation and simulation of Ambient Intelligence scenarios. Here is the link to the Internet of Things: agents can provide semantics, behavior, and autonomy to smart objects created and described with Internet of Things technology. To deal with the complexity and scalability issues that subsist in Internet of Things scenarios, we study mapping MAS models to scalable, massively parallel execution environment and hardware platforms – in this context, task allocation in MAS plays an important role, finally, in performance of the system.

This article contributes to the improvement of task scheduling for MAS based on GPGPU. Our goal is to achieve improved performance and scalability.

3 State-of-the-Art

Agent-Oriented Programming 26 (including agent-based modeling simulation 27) is a fairly young thread in software development, which extends Object-Oriented modeling / programming by concepts such as beliefs, goals and intentions [5].

Numerous methodologies for MAS development have been proposed (see 11 for an overview). FIPA 10 provides a set of standardized specifications for MAS runtime architecture and interaction protocols. The reason for it is that different problems require specific approaches. GPGPU is represented by a set of technologies, including CUDA 12, OpenCL13, and DirectCompute14. In our work, we use CUDA by NVIDIA. OpenCL is based on similar concepts, so we expect that a solution for CUDA can be rather easily ported to OpenCL.

In 15 the authors present the use of GPU technology for multiagent simulation. They demonstrate a performance improvement of their Multi-Agent Simulator by using a GPU to execute agent-based models, but they don't pay much attention to the agent environment.

In **Error! Reference source not found.** an adaptation of the FIPA standard to GPU is proposed with an example of crowd simulation. It is a continuation of earlier research 17 of the authors, where they compared JADE (Java Agent DEvelopment framework) and GPGPU, while operating with containers as in JADE. Authors claim about the necessity of standardization for agent creation on GPU.

In summary, there is a tendency in attempts to use GPU for agents, and as new GPUs are released, new devices provide more computational resources. In our research we pay great attention to agent communication, collaboration and their environment.

4 Research Contribution and Innovation

A common GPGPU application consists of two parts: host code and device code. Host code is executed by CPU, and device code is executed by GPU. Modern GPUs have a set of Streaming Multiprocessors (SM). Each SM represents a Single Instruction Multiple Threads (SIMT) architecture 18. There is a special "task manager" for every SM that groups threads, so they are finally executed in a Single Instruction Multiple Data (SIMD) manner. All threads are organized into blocks, with blocks belonging to grid(s) 19 .

In this paper, we consider three architecture variants to implement MASs that exploit GPU. The three variants are illustrated in Figure 1.

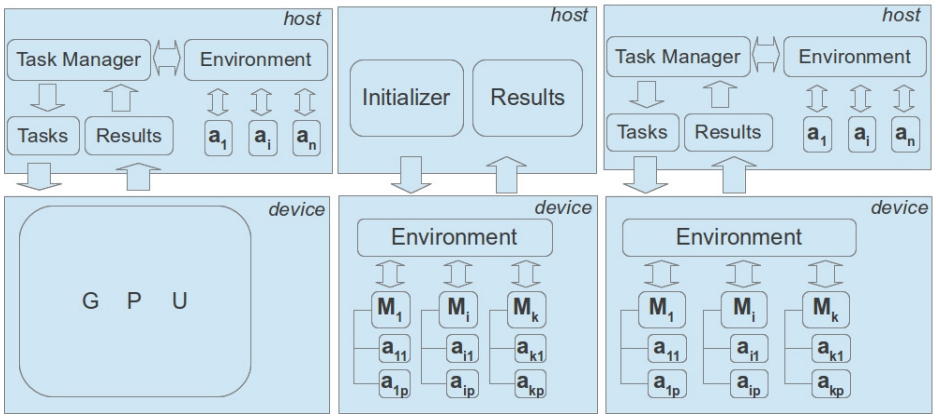

Fig. 1. Architecture variants:
a) Agents-on-Host (AoH); b) Agents-on-Device (AoD); c) Hybrid (HA)

The Agents-on-Host (AoH) model shown in Fig. 1a) consists of three top-level components: *environment*, *agents* and *task manager*. We consider the individual components in details. On the host part of the system, agents (denoted by $\mathbf{a_i}$) can communicate with each other only by means of the environment, not directly with each other. An environment here holds all information about the current system state.

The next component is Task Manager (TM) consisting of two modules: Tasks and Results. Module "Tasks" serves for GPU tasks preparation and GPU resource allocation; it also sends data to GPU and calls respective kernel functions. At the beginning, agents – which have their own tasks – make requests for computational resources to accelerate their tasks execution. Then TM starts the GPU task preparation phase, during which data obtained from agents are organized in a way that the kernel function can process them. There are numerous copies of the kernel function, each working on different data. This complies to the needs of similar agents, when they have analogous tasks. The next step is to allocate resources on the GPU and copy the data there. Then computation on the GPU is ready to start. TM sets a number of threads and number of blocks that will be executed on GPU. It means that the total number of copies of our kernel function equals the total number of GPU threads. When execution is completed, the "Results" module takes control. This module

finally synchronizes GPU threads and solutions gathered from the kernel functions back to the host. Finally, the TM provides the environment with all these solutions.

This model can be mapped to more than one machine by uniting local environments of model instances. So we will get a global environment, where agents placed on one machine can ask for computational resources on others. (Here it should be noted that characteristics of the network connection and its bandwidth need to be taken into account while evaluating the system performance).

Another model is Agents on the Device (Fig. 1b, AoD), which includes environment on GPU. Here within a GPU we consider its global memory like an environment, and kernel-function call concurrency is considerably helpful here.

The host part has two modules: *Initializer* and *Results*. *Initializer* has initial data and it also allocates GPU resources. Then it copies data to the device. It is also possible to dynamically allocate memory on GPU, but it takes special attention while doing so. For example, as we dynamically allocate GPU global memory - it is the slowest on GPU - we should evaluate the necessary amount of memory and check whether it meets the default size that we can allocate. Otherwise, it should be marked on the host before executing on the device. Another aspect of dynamic memory allocation is the expensiveness of this operation. So a good practice is to allocate memory and then just reuse it by overwriting data. The *Results* module returns the results after computation or simulation on GPU. Depending on next actions it either de-allocates GPU resources and/or passes execution to Initializer.

Components on the device are the environment, managers and agents. Every manager works within single multiprocessor and has access to the environment (here GPU global memory plays this role). As every multiprocessor has its own shared memory, which is a fast memory type on GPU, the manager is going to work with blocks. As threads running in a block can interconnect with each other and have access to shared memory, we can consider one separate thread in a block like a manager which can take part in tasks assignment for the rest threads in this block. But it makes other threads in a block waiting. After tasks assignment, this “manager” thread can also take part in computations. The main role of this manager is to synchronize threads in current block, copy data from global memory to shared one (when necessary). The amount of shared memory is limited and it should be properly used for communications between threads in this block.

After finishing all tasks on the GPU, we synchronize all GPU threads and copy data back to host. And it is Results-module that gets final results (solutions).

As an advantage, by using MAS conception, threads can be considered like agents, and every block has its manager-thread responsible for synchronization in current block and can copy necessary data to faster shared memory.

However, there are some bottlenecks in this model. As we check threads in blocks during execution and conditional expressions (if...then) are evaluated, it influences on GPU’s scheduler. It means that our threads for which these conditions are true will be executed, and others have to wait wasting time by standing idle. Also, synchronization is an expensive operation, so we use it either at the end of block execution or during threads communication and using shared memory (to protect variables in shared memory while reading/writing by threads, we use atomic functions).

And the third model – the Hybrid architecture (HA, Fig. 1c) incorporates and integrates both AoH and AoD. In the host part we have also the environment (global, which can be connected or united with remote machines), agents, and the Task Manager TM. Here agents, like in AoH model, make requests for GPU computational resources. Then, the environment passes all requests to TM. TM can act either concurrently or sequentially. It depends on the number and type of the tasks, e.g., TM can organize tasks by agent types. So the module Tasks sets final input representations for the GPU (this module is a part of TM and it also has a concurrent nature).

A device part represents the AoD model: GPU environment, manager-agents and agents (inside blocks). They act in the same manner: managers are responsible for control actions within their block, including synchronization and shared memory use. When a kernel-function completes its execution, module Results synchronizes tasks from the kernel-function and sends results back to the environment with TM help.

So our kernel-functions work independently and solve agents' tasks according to the type of agents or their tasks. TM also plays a role of load-balancer, because it should maximize the usage of both CPU and GPU to get the best performance.

5 Discussion and Critical View

We presented three types of task allocation architectures for MAS that use GPU as an underlying execution hardware. Like the authors in 20, we do not assume a strong definition of an agent, but rather take a "useful-first" approach. Conception of MAS helps to abstract from implementation details, but needs to be adapted with respect to the main restrictions of GPU. We decided to analyze the conceptual model and then implement a prototype.

The AoH model, where agents request their environment for computational resources, can be used for systems with a small number of agents. But agents can be the complex ones. Every agent should have its own tasks (which is, for example, an independent subproblem). In addition, the Task Manager ™ uses several criteria for grouping tasks, e.g. agent type, task type. For instance, if our agents are working with matrices, TM can vary available resources for concrete agents depending on the type of operation (e.g., inversion) and the amount of data.

As modern CPU is also multi-core, and we exploit this parallelism. A TM that operates with kernel-function calls puts them in separate streams and execution is asynchronous. This helps to decrease the amount of wasted computational time, but it takes a thorough analysis of the granularity: number of kernels executing in different streams.

The AoD model – with agents on GPU – suits for such tasks that could be divided into smaller subtasks that can be processed independently. For example, we should find a global minimum of a function in a space of values. We divide our space into subspaces and look for local minima. Here agents communicate when they found local minima and check whether it is less than the current minimum for all agents. After checking the whole space of values, agents will find the global minimum. To accelerate this process, a heuristic approach should be used.

The hybrid model, which is a mixture of AoH and AoD, is the most flexible among these models. Firstly, it can be scaled in two directions: locally and globally. Local scaling includes adding more GPUs to the machine and create a united GPU environment (so we have more GPU computational resources and GPU memory altogether). Global scaling implies extending environment by connecting machine environments with each other and organizing a common space. Technologies such as GPUDirect21, rCUDA22, OpenMPI23 help to implement such environments.

6 Conclusions and Future Work

In this article we presented three types of task allocation architectures for MAS which use GPU. We discussed advantages and disadvantages of every architectural type. Also we mentioned possible bottlenecks. Based on the discussion, we believe that the third (hybrid) type of model is the most promising: it is a general architecture that is aimed at performance improvement of the whole system. Moreover, there is a potential for scaling into two dimensions: local and global.

The main long-term goal of our research, of which this paper is a part, is to create a methodology of multi-agent system development with the help of general purpose computing on graphics processing units. At the end it will be realized as a framework or simulation tool. As there is a wide range of MAS application, we focus on traffic simulations and combinatorial optimization problems. Future versions could be extended to other application domains. Currently there are lots of tools for simulation, even agent-based, but the problem is that within a single machine it is difficult to achieve high performance.

Next steps are to implement all these models and create corresponding prototypes. So we can analyze performance empirically. Firstly, we will consider all models within a context of single machine, and find out the performance and behavior of the system. Then we will investigate ways of scaling, which we mentioned above.

Acknowledgments. Special thanks to Dr. Maksims Fiosins at TU Clausthal for his invaluable help.

References

1. Blatnig, S., Microscopic Traffic Simulation with Intelligent Agents: Simulation of Human Driving Behaviour, VDM (2009)
2. Rehtanz, C., Autonomous Systems and Intelligent Agents in Power System Control and Operation, Springer (2003)
3. Adler, J.L., et al, 'A multi-agent approach to cooperative traffic management and route guidance', Transportation Research, Part B, 39, 297-318 (2004)
4. Amdahl, G., Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Con.Proc.(30): 483–485 (1967)
5. Krumm, J., Advances in Ubiquitous Computing, Chapman& Hall/CRC (2009)

6. Fiosins M., Fiosina J., Müller J.P. and Görmer J. Agent-Based Integrated Decision Making for Autonomous Vehicles in Urban Traffic. In PAAMS 2011, 173-178 (2011)
7. GPGPU, www.gpgpu.org/about
8. Wooldridge, M. An Introduction to Multi-Agent Systems, 2Ed. John Wiley & Sons (2009)
9. Ehmke J. F., Fiosins M., Görmer J., Schmidt D., Schumacher H., Tchouankem H. Decision Support for Dynamic City Traffic Management Using Vehicular Communication. In Proc. of SIMULTECH' 11, SciTePress Digital Lib., pp. 327-332 (2011)
10. The Foundation for Intelligent Physical Agents, http://www.fipa.org
11. Dastani, M., Programming MAS, 5th Int. Work., ProMAS2007, USA (2007)
12. What is CUDA?, http://blogs.nvidia.com/2012/09/what-is-cuda-2/
13. OpenCL, http://www.khronos.org/opencl
14. Direct Compute, https://developer.nvidia.com/directcompute
15. Laville, G. et al, Using GPU for Multi-Agent Multi-Scale Simulations, DCAI 2012Vol. 151, pp 197-204, Springer, Berlin, Heidelberg (2012)
16. Oliveira, L.G., Gonzales, E.W., Bernardini, F. C., A Parallel Fipa Architecture Based on GPU for Games and Real Time Simulations, LNCS Vol. 7522, pp 306-317 (2012)
17. Luiz Guilherme Oliveira dos Santos et al, Mapping Multi-Agent Systems Based on FIPA Specification to GPU Architectures, VIDEOJOGOS 2010 (2010)
18. SIMT,http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture
19. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model
20. Johnson, T., Rankin, J., Parallel Agent systems on a GPU for use with Simulations and Games, 12th WSEAS ACS '12 (2012)
21. GPUDirect, developer.nvidia.com/gpudirect
22. rCUDA, www.rcuda.net
23. OpenMPI, www.open-mpi.org
24. Kallmann, M., Thalmann, D., Modeling Objects for Interaction Tasks, 9th EGCAS, Lisbon, Portugal, 73-86 (1998)
25. Smith, R.G.. 1980. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Comput.* 29, 12 (December 1980), 1104-1113.
26. Shoham,Y., Agent-oriented programming. *Artif. Intell.* 60, 1 , 51-92 (March, 1993).
27. F. Klügl, A.L.C. Bazzan: Agent-Based Modeling and Simulation. AI Magazine 33(3): 29-40 (2012)
28. Fischer K., Müller J.P., Pischel, M., Cooperative Transportation Scheduling: an Application Domain for DAI. Journal of Applied Artificial Intelligence, 10, 1-33 (1996).