Running Reinforcement Learning Agents on GPU for Many Simulations of Two-Person Simultaneous Games

Koichi Moriyama*‡, Yoshiya Kurogi*, Atsuko Mutoh*, Tohgoroh Matsui†, and Nobuhiro Inuzuka*
*Department of Computer Science, Nagoya Institute of Technology, Nagoya, Japan
†Department of Clinical Engineering, Chubu University, Kasugai, Aichi, Japan
‡Email: moriyama.koichi@nitech.ac.jp

***Abstract*—It is desirable for multi-agent simulation to be run in parallel; if many agents run simultaneously, the total run time is reduced. It is popular to use GPGPU technology as an inexpensive parallelizing approach in simulation, but the "agents" runnable on GPU were simple, rule-based ones like elements in a scientific simulation. This work implements more complicated, learning agents on GPU. We consider an environment where many reinforcement learning agents learning their behavior in an iterated two-person simultaneous game while changing peers. It is necessary to run many simulations in each of which a pair of agents play the game. In this work, we implement on GPU the simulations where the agents learn with reinforcement learning and compare two methods assigning the simulations to GPU cores.**

***Index Terms*—Reinforcement Learning, Multi-agent Based Simulation, GPGPU, High Performance Computing.**

1. Introduction

In multi-agent simulations, computation time is increasing as the number of agents is increasing. Researchers are trying to accelerate multi-agent simulations by running in parallel on a computer cluster that consists of many computers connected with a high speed network. The idea is simple; if many agents can run simultaneously, the total run time can be reduced. However, it is too expensive to be widely used.

On the other hand, general purpose GPU (GPGPU) technology has been widely used as an inexpensive parallelizing approach. It uses graphics processing units (GPUs) apt at parallel computing for general purpose computation. It is more affordable than computer clusters, but the "agents" runnable on GPU were simple, rule-based ones like elements in a scientific simulation.

This work investigates GPGPU technology in a multi-agent simulation where more complicated agents run on GPU. We consider an environment where many reinforcement learning (RL) agents learn their behavior in an iterated two-person simultaneous game while changing peers. The agents have their own *appraisal mechanisms* that will evolve using an accumulated payoff after playing the game with all other agents [1], [2]. In this work, we investigate how the accumulated payoff is efficiently obtained by running simulations on GPU where the agents learn with RL, and propose two ways how we assign the simulations to GPU cores.

This work was partly supported by JSPS KAKENHI Grant Number JP16K00302 and JP19K12118, the Hori Sciences & Arts Foundation, and Kayamori Foundation of Informational Science Advancement.
Y. Kurogi is currently with Capcom Co. Ltd. Tokyo Branch, Tokyo, Japan after his graduation. Capcom is not related to this work at all.

2. Background

2.1. Multi-agent Based Simulation

Multi-agent based simulation (MABS) models and simulates a target environment by behaviors of *individuals* [3], [4]. It is compared to *object-oriented* simulation (OOS), but there is no clear border between them. Davidsson [3] discussed the difference between them using six dimensions including *proactiveness* and *adaptivity*. In OOS, individuals are reactive and static, which means that their behaviors are controlled by fixed rules. On the other hand, MABS mainly targets more complex systems like (collective) human behaviors [4] that are proactive and temporally variable due to learning and adaptation of individuals.

Since one of main problems of simulation is scalability, researchers want to run it in parallel. GPGPU technology has been used in both OOS and MABS. In the early days of using GPU in MABS, it was used for running agents on GPU in a domain-specific simulation, *e.g.*, [5], [6]. However, agents in the simulation were reactive and static like objects in OOS. In particular, the adaptivity was not considered at all.

Generic MABS on GPU is challenging due to difficulties such as requiring special programming skills. In fact, there are only several generic parallel MABS platforms all of which run on computer clusters [7]. Nowadays, if we use GPU to run MABS with complicated agents, it is recommended that it is run on a hybrid system where only the environment is on GPU while the agents are on CPU [8].

2.2. GPU and GPGPU

GPU is a processor dedicated to real-time image processing that calculates hundreds of thousands vertices and changes colors of millions of pixels on the screen simultaneously. GPU has to do (relatively) simple calculations many times; therefore it has much more computing cores[1] than CPU and runs them in parallel.

GPU uses SIMT (single instruction, multiple threads) architecture that processes multiple threads with a single instruction. The number of threads sharing an instruction, called *warp* or *wavefront*, is specified by the GPU architecture. It has a major weakness: If there is a conditional statement that changes a process with respect to each status, every core has to run all codes in the statement and choose a result corresponding to the status. It results in increase of total run time; hence the programs running on GPU should have few conditional/loop statements.

The parallel computing performance of GPU can be used in other computation as well as graphics. In the late 2000s, it became common that GPU was used for general purpose computation. It is because GPGPU computation frameworks were released. There are two major frameworks: CUDA[2] and OpenCL[3]. CUDA works only on GPUs of NVIDIA corporation, but OpenCL works on many kinds of processors including AMD GPUs and Intel onboard GPUs as well as NVIDIA's. Due to its flexibility, this work uses OpenCL.

OpenCL uses an n-dimensional thread space where $1 \leq n \leq 3$. *Work-item* is the smallest unit in the thread space where a GPU program called *kernel* runs. We call a work-item a *thread* hereafter. *Work-group* is a unit consisting of several work-items running simultaneously sharing a memory space.

3. Simulation Target

This work implements on GPU a multi-agent simulation where RL agents run. The environment used here is that in our previous works where many RL agents learned their behavior in an iterated prisoner's dilemma (PD) game while changing peers. We proposed *utility-based Q-learning agents* [9] that did Q-learning [10] using *subjective utilities* instead of given rewards, which were derived from the rewards using an *appraisal mechanism* of the agent. After that we investigated what appraisal mechanisms appeared by evolutionary computation using genetic algorithm (GA) in the environment [1], [2].

3.1. Prisoner's Dilemma Game

A PD game [11], [12] is a two-person two-action simultaneous game shown by a payoff matrix (Table 1). Two players called "Row" and "Column" simultaneously choose their actions from rows and columns of the matrix, respectively. The two actions are respectively called C (cooperation) and D (defection). After choosing his/her action, each player obtains a payoff T, R, P, or S in the matrix. For example, when Row chooses C and Column chooses D, they obtain payoffs S and T, respectively.

1. Precisely, they should be called "processing elements", but we use the term "core(s)" instead in this paper.
2. https://developer.nvidia.com/cuda-toolkit
3. https://www.khronos.org/opencl/

TABLE 1. PRISONER'S DILEMMA PAYOFFS

Row \ Column	C	D
C	R, R	S, T
D	T, S	P, P

PD has the following relations among the payoffs: $T > R > P > S$. Under these relations, each player obtains a larger payoff whenever choosing D regardless of the opponent's action. As a result, both players choose D and obtain P. However, it is more desirable for them to choose C mutually and obtain $R (> P)$.

3.2. Utility-Based Q-learning

A Q-learning agent [10] has an action value function Q showing an estimated expected return afterward. Q is updated by the following rule using the current and the next states $s_t, s_{t+1} \in \mathcal{S}$, the action $a_t \in \mathcal{A}(s_t)$, and the given reward r_{t+1}. Note that $\mathcal{S}$ is a set of possible states, $\mathcal{A}(s)$ is a set of available actions in a state s, and α, γ are parameters.

$$Q_{t+1}(s, a) = \begin{cases} Q_t(s_t, a_t) + \alpha\delta_t & \text{if } s = s_t, a = a_t, \\ Q_t(s, a) & \text{otherwise,} \end{cases} \tag{1}$$

$$\delta_t \equiv r_{t+1} + \gamma \max_{a' \in \mathcal{A}(s_{t+1})} Q_t(s_{t+1}, a') - Q_t(s_t, a_t). \tag{2}$$

A utility-based Q-learning agent [9] internally has an appraisal mechanism that derives *subjective utilities* from given rewards, and uses them as rewards of Q-learning. In other words, δ_t is changed to the following:

$$\delta_t \equiv u(r_{t+1}) + \gamma \max_{a' \in \mathcal{A}(s_{t+1})} Q_t(s_{t+1}, a') - Q_t(s_t, a_t), \tag{3}$$

where $u(r_{t+1})$ is the subjective utility derived from a function of given reward r_{t+1}. Note that the utility-deriving function u is arbitrary and may depend on other variables as well as the immediate reward r_{t+1}.

3.3. Evolution of Utility-Deriving Functions

Since the utility-deriving function u is arbitrary, we investigated what functions appeared from evolutionary computation [1], [2]. We assumed that the utility-deriving function was a cubic function of the payoff r:

$$u(r) \equiv ar^3 + br^2 + cr + d,$$

and evolved the real number coefficients a, b, c, d with the simple GA [13] where the fitness was the sum of received payoffs (*i.e.*, not utilities) of the agent.

The simulation algorithm was as follows:

1) Generate N utility-based Q-learning agents from N chromosomes showing the coefficients created randomly.
2) For all pairs of N agents, they play a PD game and update their action-value functions M times.
3) Rank all agents by their fitness.
4) Run the GA to generate N next-generation agents based on the ranking.
5) Unless the end condition is satisfied, back to 2.

Note that, in the process 2, the action-value function each agent had was initialized when the pair was changed, *i.e.*, every pair was independent.

In the process 4, to generate two next-generation agents, the GA chose two agents, exchanged genes in their chromosomes, and mutated genes in the chromosomes. There were two parameters: the crossover probability p_c and the mutation probability p_m. p_c determined whether the crossover process would start or not. If satisfied, each gene in the two chromosomes was exchanged one by one randomly. p_m determined whether each gene would be mutated or not. If satisfied, $x \sim N(\mu, \sigma)$ was added to the gene.

4. Implementation

When we run the simulation, the process 2 has to be run $O(N^2)$ times in a generation. As in the previous work [1], when we set $N = 100$, $M = 1000$, and as the end condition of process 5 the number of generation G to 10000, it takes more than 20 minutes with a modern PC server. Note that it gives us only a sample of one simulation with a certain parameter set. Usually, we have to run many simulations to discuss the result statistically and know the effect of parameters. Since the number of combination of parameters is huge, the whole simulation time easily becomes several hours, days, or weeks. It is why we need parallel computation.

The main problem of parallel computation is how we divide the whole problem into independently runnable parts. Since the computation time is almost in evaluation of the individuals, and the target environment can easily be divided because every pair is independent, we use the master-slave type parallel model of GA [14]. In the model, one processor, called *master*, is responsible for GA operations and control of the whole process, while other processors, called *slaves*, are responsible for evaluation of individuals. First, the master assigns all individuals to the slaves and orders every slave to evaluate the assigned individuals and return their fitness values to the master. After receiving fitness values of all individuals, the master conducts the genetic operations to the individuals to generate the next generation. Note that, since the master-slave model only divides the evaluation of individuals into many processors, it gives us the same result of original GA [14].

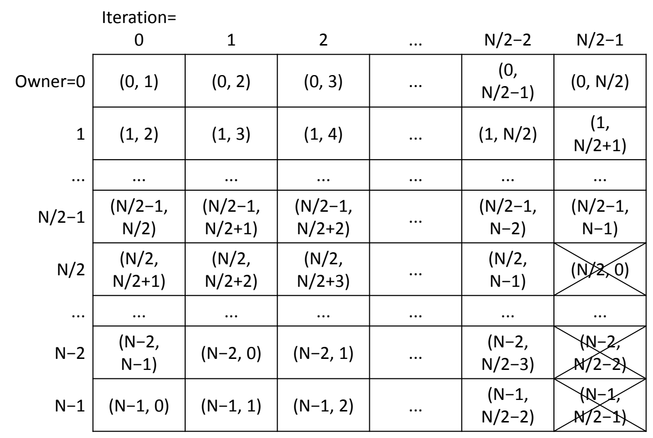
GPU has thousands of computing cores, which is much more than CPU. However, the computation speed of GPU cores is much less than that of CPU cores. Parallelization is not free; even if no communication among threads is needed, creating threads may take non-negligible time. If there is no processor remaining, threads have to wait for a while until a processor will be available. Based on the discussion, this work considers the following two assigning methods having different levels of parallelization.

4.1. Agent-Based Assignment

Agent-Based Assignment (AA) creates one thread for an agent, *i.e.*, the number of threads are that of agents. The thread ID is equal to the ID of agent who *owns* this thread. After creating these threads, every thread runs the process 2 of Section 3.3 $\binom{N}{2}/N = (N-1)/2$ times in average.

Simply let us use a 1-dimensional thread space. As we saw in Section 2.2, every thread joins in a work-group. It is known that the number of threads in a work-group, or *work-group size*, is good to be set to a multiple of the warp size, but we want to minimize the work-group size because the overhead of synchronization after processing and that of conditional/loop statements may be smaller as the size is smaller. The threads in this implementation do not need information sharing nor synchronization during processing. Then, we set the work-group size is equal to the warp size. Note that, the total number of threads should be a multiple of the work-group size. Therefore, we create $\lceil N/w \rceil \times w$ threads where w is the warp size.

In each evaluation process, the owner agent whose ID is T plays with the opponent whose ID is $(T + c) \bmod N$ where c is a counter starting from 1 to $\lceil (N-1)/2 \rceil$. If N is even, the owner whose ID is $\geq N/2$ stops running the last iteration because the last pair has already been run in another thread whose owner is the opponent (Fig. 1).



	Iteration= 0	1	2	…	N/2−2	N/2−1
Owner=0	(0, 1)	(0, 2)	(0, 3)	…	(0, N/2−1)	(0, N/2)
1	(1, 2)	(1, 3)	(1, 4)	…	(1, N/2)	(1, N/2+1)
…	…	…	…	…	…	…
N/2−1	(N/2−1, N/2)	(N/2−1, N/2+1)	(N/2−1, N/2+2)	…	(N/2−1, N−2)	(N/2−1, N−1)
N/2	(N/2, N/2+1)	(N/2, N/2+2)	(N/2, N/2+3)	…	(N/2, N−1)	~~(N/2, 0)~~
…	…	…	…	…	…	…
N−2	(N−2, N−1)	(N−2, 0)	(N−2, 1)	…	(N−2, N/2−3)	~~(N−2, N/2−2)~~
N−1	(N−1, 0)	(N−1, 1)	(N−1, 2)	…	(N−1, N/2−2)	~~(N−1, N/2−1)~~

Figure 1. Agent-based Assignment, when N is even. Each row and column correspond to a thread and the number of iteration, respectively. Each thread runs $N/2$ times, but the cells with crosses indicate these pairs are run in another thread. For example, the last iteration of owner $N/2$ is run in the last iteration of owner 0.

4.2. Pair-Based Assignment

Pair-based assignment (PA) creates one thread for each pair, *i.e.*, the number of threads are that of pairs in total.

Each thread has an ID that determines which pair is assigned to the thread. After creating these threads, the evaluation process 2 in Section 3.3 is run in parallel.

The number of threads becomes $\binom{N}{2} = N(N-1)/2$. Simply let us use a 1-dimensional thread space. From the same reason of the AA case, we create $\lceil N(N-1)/2w \rceil \times w$ threads where w is the warp size.

Next we have to assign a pair to each thread. It is not trivial to derive a pair from a thread ID T in the 1-dimensional space. All pairs needed to be assigned are shown in Fig. 2, where N is even for example. We have to assign a thread to every pair, but it is difficult to calculate a pair directly from T. Admittedly we can naively assign a thread one by one from $(0, 1)$ to $(N-2, N-1)$, but it costs due to loop statements.

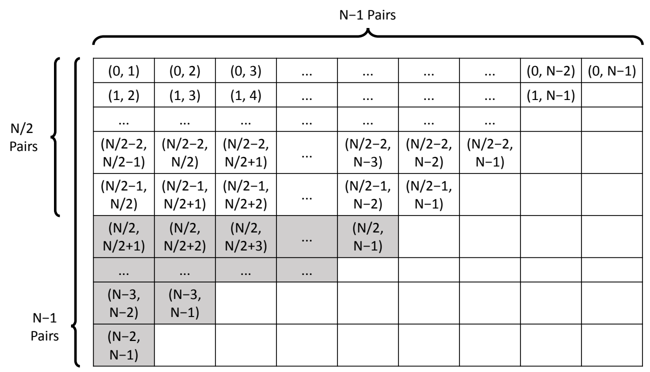
The difficulty is in the fact that different rows have different numbers of pairs. It is important to note that we have to *enumerate* all pairs but the order of the pairs is not the matter. Therefore we move some pairs and make it easy to calculate a pair from T. The result is shown in Fig. 3. The gray cells in Fig. 2 are moved to fill the empty cells and consequently all but the last row have N pairs. As we can see that the row number corresponds to one agent of a pair in a white cell while the column number corresponds to one agent of a pair in a gray cell. We can do similarly when N is odd.

This process is generally described as follows. First we calculate the coordinate of a cell (M_x, M_y) from T:

$$M_x \equiv \lfloor T/N \rfloor,$$
$$M_y \equiv T \bmod N.$$

Then, we assign to this thread two agents whose IDs are

$$\begin{cases} (M_x,\ M_x + M_y + 1) & \text{if } M_x + M_y + 1 < N \\ ((N-2) - M_x,\ M_y) & \text{otherwise.} \end{cases}$$

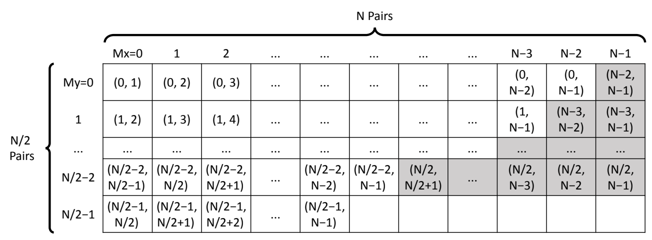

N−1 Pairs
N/2 Pairs
N−1 Pairs

(0, 1)	(0, 2)	(0, 3)	...	...	...	...	(0, N−2)	(0, N−1)
(1, 2)	(1, 3)	(1, 4)	...	...	...	...	(1, N−1)	
...	...	...	...	...	...	...		
(N/2−2, N/2−1)	(N/2−2, N/2)	(N/2−2, N/2+1)	...	(N/2−2, N−3)	(N/2−2, N−2)	(N/2−2, N−1)		
(N/2−1, N/2)	(N/2−1, N/2+1)	(N/2−1, N/2+2)	...	(N/2−1, N−2)	(N/2−1, N−1)			
(N/2, N/2+1)	(N/2, N/2+2)	(N/2, N/2+3)	...	(N/2, N−1)				
...	...	...	...					
(N−3, N−2)	(N−3, N−1)							
(N−2, N−1)								

Figure 2. All pairs that will be assigned to threads, when N is even. Each number is an ID of an agent, from 0 to $N-1$.

5. Experiment

This section evaluates the use of GPGPU technology for many simulations of multiple RL agents using the above two assignment methods.


N Pairs
N/2 Pairs

	Mx=0	1	2	...	...	...	...	...	N−3	N−2	N−1
My=0	(0, 1)	(0, 2)	(0, 3)	...	...	...	...	...	(0, N−2)	(0, N−1)	(N−2, N−1)
1	(1, 2)	(1, 3)	(1, 4)	...	...	...	...	...	(1, N−1)	(N−3, N−2)	(N−3, N−1)
...	...	...	...	...	...	...	...	...	...	...	...
N/2−2	(N/2−2, N/2−1)	(N/2−2, N/2)	(N/2−2, N/2+1)	...	(N/2−2, N−2)	(N/2−2, N−1)	(N/2, N/2+1)	...	(N/2, N−3)	(N/2, N−2	(N/2, N−1)
N/2−1	(N/2−1, N/2)	(N/2−1, N/2+1)	(N/2−1, N/2+2)	...	(N/2−1, N−1)						

Figure 3. After moving pairs to empty cells. M_x and M_y indicate the coordinates of cells.

The computer used has two Intel Xeon E5-2650 v4 CPU (24 physical / 48 hyperthreading cores in total) and one GeForce GTX 1080 GPU (2560 cores). The warp size of GTX 1080 is 32.

GPU programs are implemented in OpenCL framework, version 1.2. The host programs are written in C++14, compiled by GNU GCC 7.2.1 with compile options "`-O3 -march=native`". The kernel program is written in OpenCL C. Note that, since no random number generator is in OpenCL C specification, we implemented Xorshift 128 bit algorithm in the kernel program. In the host programs, we used `std::mt19937`, a mersenne twister engine.

We implemented the following programs: Naive, GPU-AA, GPU-PA, CPU-AA, and CPU-PA. *Naive* is a program running on a single thread of CPU. *GPU-AA* and *GPU-PA* are those explained in Section 4. *CPU-AA* and *CPU-PA* are similar to *GPU-AA* and *GPU-PA*, respectively, but they run on CPU threads instead of GPU. Note that it is known that C++ standard thread implementation of GNU GCC always creates new threads instead of using a thread pool where old threads were stored and reused, while that of some other compilers like Microsoft C++ uses a thread pool. Since it takes time to create a new thread, it is unfair for comparison. Therefore we used an implementation of thread pool instead of standard threads in CPU-AA and CPU-PA.

All parameters used in the experiment are identical with the previous work [1], if not otherwise specified. The number of agents N is 100, the number of iterations of PD game M is 1000, and the number of generations G is 10000. The payoffs of PD game are as follows: $T = 5$, $R = 3$, $P = 1$, and $S = 0$, which satisfy the PD condition. The utility-based Q-learning parameters are as follows: $\alpha = 0.25$ and $\gamma = 0.5$. For action selection, the agent uses ϵ-greedy action selection method that chooses a random action with probability ϵ, which is set to 0.05. The crossover and mutation probabilities are set as $p_c = 0.9$ and $p_m = 0.01$. The mutation value x is derived from $N(0, 1)$.

Note that OpenCL compiles kernels from source files on the fly, but the compiled kernels are stored as cache files. We disabled the cache mechanism because we want to time the whole process from scratch.

5.1. Validity of the Programs

First of all, we confirm that the programs were properly implemented. Figure 4 shows histograms of average payoffs of the last generation in 100 runs of each program. We can see that all programs gave similar results having two peaks around 1.4 and 2.8. Their heights were also similar. Based on the results, we conclude that the programs were valid.

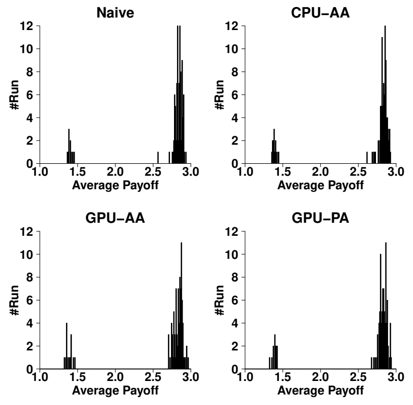


Figure 4. Histograms of average payoffs of the last generation in 100 runs of each program. X-axis shows bins of average payoffs whose width is 0.01, while y-axis shows the number of runs within the bin.

5.2. Run Time

This section compares the programs by the elapsed time of the above experiment. Table 2 shows the results that are the average of 100 runs. Naive needed more than 20 minutes, but CPU-AA and CPU-PA needed about two and six minutes, respectively. GPU-PA was the fastest; it needed only 20 seconds, but GPU-AA was slower than CPU threads.

TABLE 2. ELAPSED TIME FOR 100 RUNS, WHEN $N = 100$, $M = 1000$, AND $G = 10000$.

Program	Elapsed time (sec.)
Naive	1279.67
CPU-AA	134.14
CPU-PA	246.11
GPU-AA	591.27
GPU-PA	19.82

5.3. Scalability

We see another experiment where the number of agents N was increased to 200, 300, ..., 20000. We set $G = 1$ because we only wanted to know the effect of the number of agents on the evaluation time. Note that the evaluation was done twice, before and after genetic operations. Also, since we wanted to know only the processing time, outputs for consoles and files were stopped.

Figures 5 and 6 depict the relation between the number of agents and the elapsed time averaged by 10 runs. GPU needed setup time about one second in every run while CPU did not. Therefore when the number of agents was small GPU took more time than CPU. However, after that the number increased more than 700 for PA and 1500 for AA, GPU was faster than CPU.

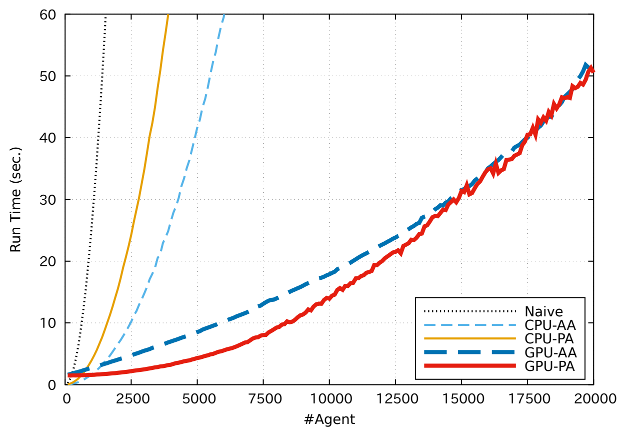


Figure 5. Scalability: The relation between the number of agents and the elapsed time averaged by 10 runs, when $G = 1$.

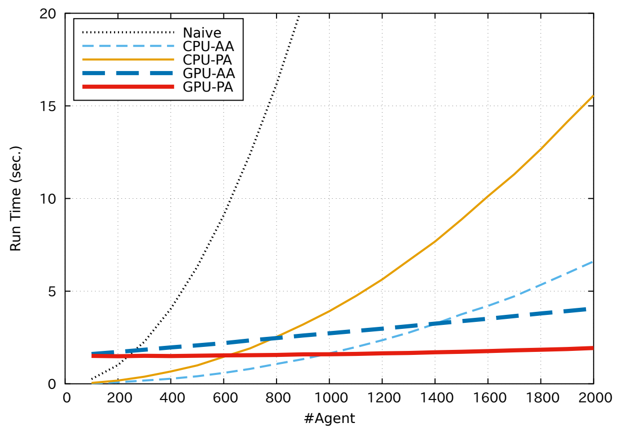


Figure 6. Closeup view of Fig. 5.

6. Discussion

Run-time results showed that the programs using multiple threads were obviously faster than the naive one. First let us see details of these results.

CPU-AA was faster than CPU-PA. Intuitively the pair-based method seemed faster than the agent-based method due to the degree of parallelization; but it was not. It may be because the CPU cores including hyperthreading on the machine were only 48, which is about a half of $N = 100$. That is, both programs fully used CPU cores. The difference is that CPU-PA created much more threads ($N(N-1)/2 = 4950$) than CPU-AA ($N = 100$). As mentioned before, it costs to create threads. This result shows that the cost is larger than that of iterations in each thread.

In addition, GPU-AA was 4.4 times slower than CPU-AA unexpectedly. It is due to the calculation speed of cores. CPU-AA used 48 CPU cores, which is about a half of GPU cores GPU-AA used ($N = 100$). That is, roughly speaking, a GPU core takes about $4.4 \times 2 = 8.8$ times longer than a CPU core; it is plausible.

Next let us see details of results of the scalability experiment. CPU-AA and CPU-PA were clearly faster than Naive, while CPU-AA was faster than CPU-PA from beginning to end. It is, as we saw, due to the cost for creating threads. The result showed that CPU-PA took about 2.4 times more time than CPU-AA regardless of the number of agents. That is, creating threads costed 2.4 times more than iteration in this environment. This cost will be mitigated when the number of generations G become large, as in the run-time results ($246.11/134.14 \approx 1.83$); this is probably due to the thread pool. On the other hand, given that CPU-PA was faster than Naive, the best number of threads probably exists. The authors speculate that it is 24 or 48, *i.e.*, the number of physical cores or that of hyperthreading cores, respectively.

GPU programs were much more scalable than CPU ones. GPU-PA was faster than GPU-AA at the beginning, but their performance became similar at the end. The reason GPU-PA was faster than GPU-AA when $N \leq 2500$ is obvious; GPU-PA used all GPU cores while GPU-AA did not. Also, it is known that GPU performs better if running threads are more than its cores because when one thread has to read a memory, GPU can allow another waiting thread to run its program. Since memory access usually takes much longer time than computing instructions, the best performance is given when the number of threads is several times of that of cores. This is probably why the performance of GPU-PA and that of GPU-AA became similar at the end; both programs fully used the GPU at that time. When there are 15000 agents, GPU-AA uses 15000 threads that is about 6 times of the cores.

There remains a question. It is not clear why the GPU-PA result fluctuated in $N \geq 8000$. When $N = 8000$, the number of threads was about 32 millions. The fluctuation may be caused by such too many threads, but it is difficult for us to answer the question. It may depend on scheduling and memory usage in the GPU we cannot know easily.

7. Conclusion

This work investigated the use of GPGPU technology in MABS. In particular, we implemented a simulation where RL agents played a PD game and learned their behaviors iteratively, while their appraisal mechanisms evolved from accumulated payoffs. In this work, GPU was used in calculation of fitness of the appraisal mechanisms of agents, *i.e.*, in the process where all agents played a PD game with all of the others and learned their behavior with RL. Note that it is more complicated than rule-based agents GPU has been used in MABS.

We introduced two assignment methods called AA, the agent-based assignment, and PA, the pair-based assignment. Intuitively, PA would effectively utilize GPU's parallel computation performance based on many cores. Indeed PA on GPU obtained the best performance among five programs; however, the performance was not different from AA when the number of agents was huge. It is probably both methods fully used the GPU in this case.

The environment we considered is only the PD game, but other two-person simultaneous games can similarly be used. The number of actions each agent chooses affects the amount of memory because each agent has to remember more values of actions and/or states. It may affect the number of threads GPU can run simultaneously. However, it is a problem of RL and may be mitigated by memory-saving methods like function approximation.

On the other hand, the number of players in the game critically affects the result because it changes the number of combination of agents drastically. The two proposed methods differ in the level of parallelization. Thus we will discuss the appropriate level of parallelization in accordance with the number of players in a game in the future.

References

[1] K. Moriyama, S. Kurihara, and M. Numao, "Evolving Subjective Utilities: Prisoner's Dilemma Game Examples," in *Proc. 10th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2011, pp. 233–240.

[2] M. Miyawaki, K. Moriyama, A. Mutoh, T. Matsui, and N. Inuzuka, "Evolution Direction of Reward Appraisal in Reinforcement Learning Agents," in *Proc. 12th KES Symposium on Agent and Multi-Agent Systems – Technologies and Applications (KES-AMSTA)*, 2018, pp. 13–22.

[3] P. Davidsson, "Multi Agent Based Simulation: Beyond Social Simulation," in *Multi-Agent-Based Simulation*, S. Moss and P. Davidsson, Eds. Springer, 2000, pp. 97–107.

[4] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," *Proceedings of the National Academy of Sciences*, vol. 99, pp. 7280–7287, 2002.

[5] K. S. Perumalla and B. G. Aaby, "Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs," in *Proc. 2008 Spring Simulation Multi-Conference (SpringSim)*, 2008, pp. 116–123.

[6] U. Erra, B. Frola, V. Scarano, and I. Couzin, "An efficient GPU implementation for large scale individual-based simulation of collective behavior," in *Proc. 2009 International Workshop on High Performance Computational Systems Biology (HIBI)*, 2009, pp. 51–58.

[7] A. Rousset, B. Herrmann, C. Lang, and L. Philippe, "A survey on parallel and distributed multi-agent systems for high performance computing simulations," *Computer Science Review*, vol. 22, pp. 27–46, 2016.

[8] E. Hermellin and F. Michel, "GPU Delegation: Toward a Generic Approach for Developping MABS using GPU Programming," in *Proc. 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2016, pp. 1249–1258.

[9] K. Moriyama, "Utility based Q-learning to facilitate cooperation in Prisoner's Dilemma games," *Web Intelligence and Agent Systems*, vol. 7, no. 3, pp. 233–242, 2009.

[10] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[11] R. Axelrod, *The Evolution of Cooperation*. Basic Books, 1984.

[12] W. Poundstone, *Prisoner's Dilemma*. Doubleday, 1992.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[14] E. Cantú-Paz, "A Survey of Parallel Genetic Algorithms," *Réseaux et systèmes répartis, calculateurs parallèles*, vol. 10, no. 2, pp. 141–171, 1998.