Parallel Agent systems on a GPU for use with Simulations and Games

TIMOTHY JOHNSON
La Trobe University
Computer Science and Engineering
3086 Plenty Road Bundoora VIC
AUSTRALIA
twjohnson@students.latrobe.edu.au

JOHN RANKIN
La Trobe University
Computer Science and Engineering
3086 Plenty Road Bundoora VIC
AUSTRALIA
j.rankin@latrobe.edu.au

Abstract: In this paper we describe a parallel agent based computing system. The agents are placed on GPU memory and executed in parallel on the GPU. We discuss the difficulties in creating this system and provide solutions to each of the problems encountered. We then go on to describe a test bed application for the implementation of our ideas. Following this, we test our application sand discuss the results before making conclusions about the viability of such a system for use beyond research. Finally, we discuss future work that can be performed to enhance the system.

Key–Words: Agents, GPGPU, Simulation, Games

1 Introduction

In this section we'll introduce agents and GPGPU. We'll define these two concepts and explain how and where they have been used in past research.

In the next section, we'll talk about the concept of combining agents with GPGPU and the motivation behind doing this. We'll also go into depth talking about the problems faced during this combination process and the proposed solutions to each of these problems.

In the third section, we'll talk about a test bed application developed using C++, OpenCL and OpenGL. We'll discuss the design of the system and of the agents that live inside the system. Details will be provided about the implementation of each of the solutions proposed in section two.

The fourth section will test the system described in section three. Testing will occur both on different pieces of hardware and with differing numbers of agents to test the scalability and speed up values of the system. We'll also discuss these results and their implications.

Section five will provide a conclusion about the results collected in section four.

The last section, section six, will discuss future work that can be done to enhance the system.

1.1 Agents

Agents are an interesting case in the realm of software research and development. Lacking in a standard, formal definition, anything said about what exactly an agent is very much left open to interpretation. Rather than trying to say what we think an agent is, we will attempt to offer an explanation as to what an agent should be able to do. Agents are capable of independent, autonomous action[10], working to achieve a goal on someone's (possibly their own) behalf. Agents are also capable of interacting with other agents or humans, cooperating and negotiating as they see fit. Some agents are also able to learn from experience.

On a more technical level an agents inner workings can be thought of as a continuous loop made up of the following four key steps:

- Observation. Observe the world and update any beliefs we (the agent) have about the world.
- Deliberation. Decide what we need to do with this information, what should our intention or goal be?
- Planning. With a set intention, how can we do accomplish this goal? Generate a plan to achieve our goal.
- Execution. Execute the plan generated in the previous step.

During runtime, our agent also needs to take other things into consideration. For example, when should we perform deliberation again? If we don't deliberate often enough, the plan we concocted may no longer be relevant by the time we complete it, and conversely, if we deliberate too much we way never actually achieve anything. We can consider an agent to be reactive or proactive depending on how much time it spends de-

ISBN: 978-1-61804-092-3

liberating about the changes in its environment.

Agents generally exist in multi-agent systems where they must communicate, bargain and cooperate with other agents to achieve their goals. Different agents in a multi-agent system have their own 'spheres of influence' which can coincide with the spheres of other agents, resulting in interesting interactions between these agents. We should keep in mind that each agent in a multi-agent system is self interested and will behave as such.

With the explosion in usage of the internet over the last few years we have seen the value of computerised agents increase drastically, especially in the areas of e-commerce[6] and matching services[7] as people work to exploit the riches of on-line markets. Agents also continue to find uses for themselves in the area of computer simulation. For example, they have been used to model pedestrian dynamics[9] in work done by Toyama, Bazzan and da Silva. In addition the games industry can also benefit from the usage of agents in the form of bots handling the artificial intelligence vital to the success of so many modern video games[3].

1.2 GPGPU

The power of Graphical Processing Units (GPUs) has been increasing drastically over the last few years, pushed along by the growth of the gaming industry and the resulting demand for cheap, powerful hardware. In the past GPUs contained both pixel shaders and vertex processors[5]. However, this caused a problem: how do you balance the amount of pixel shaders to vertex processors? Going too heavy either way would result in wasted hardware during processing. The solution to this problem was to create general processing units that could perform both tasks, as well as other work. And thus, General Purpose GPU (GPGPU) computing was born.

The strength of GPUs as processing units lies in their structure. A GPU consists of many Multi-Processors (MPs), each of which has many Stream Processors (SPs). Each of the SPs executes in SIMD (single instruction, multiple data) mode. Specifically, each of the processors executes the same instructions at the same time, but does so on different parts of memory. On a GPU the memory can be roughly divided into three different groups. Firstly, the global memory is accessible by all processors, but has high latency, making access to it slow. Secondly, the shared memory, which is memory specific to a single MP and only accessible to the SPs on that MP. This memory has the advantage of being quite low latency allowing for fast access times to it. And finally the private memory which is accessible only by the SP it is attached to. Illustration figure 1 can help to visualise the structure of a GPU.

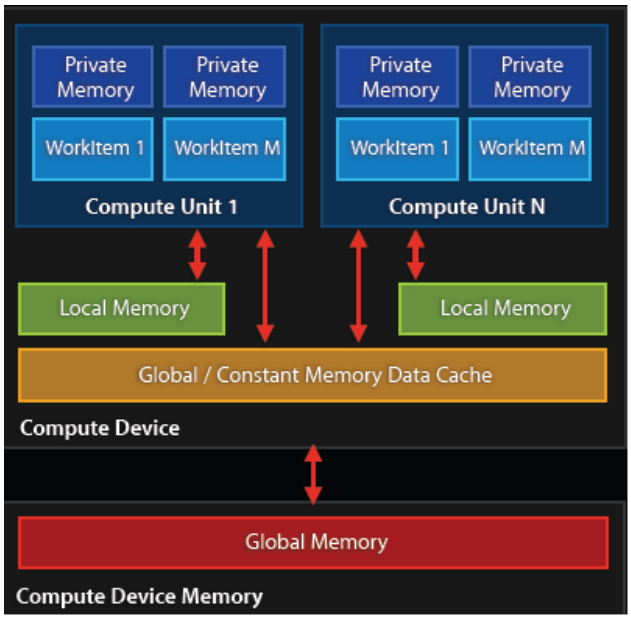


Figure 1: GPU Memory Structure[1]

The potential processing power of this massively parallel structure has drawn the interest of quite a few researchers in fields extending far beyond that of video game development. The areas of physics, cosmology[8], chemistry, evolutionary algorithms[5], mathematics and even the search for extraterrestrial life, just to name a few, have all found themselves greatly benefiting from a boost in processing power provided by GPUs. With Nvidia's CUDA, AMD's Stream and the multi-platform OpenCL making GPGPU accessible to the general public we can expect more and more people to find themselves looking to exploit the power of the GPU. And as the power of GPUs grows, we need to continue to find more ways to make use of this immense potential.

2 Combining Agents and GPGPU

Agents are able to work independently as well as cooperate with one another to achieve their goals. GPUs are excellent at processing large amounts of data in parallel. Is there anyway we can combine these two things? And what would be our motivation for doing this in the first place?

Forty years ago when video games first started to hit the mass markets in the arcades and homes, the artificial intelligence (AI) they included could only be classified as poor. Games released today still have poor AI. Graphics, sounds, gameplay, story-telling have all been able to benefit from improvements in

ISBN: 978-1-61804-092-3

hardware, so why not AI? Game AI today, in most cases, can't really be thought of intelligent at all. For the most part it consists of preprogrammed scripts which follow the same set of instructions over and over, with no actual thinking involved.

Now, before we continue we should make the important clarification that agents are not AI. What agents give us however, is a way of accessing AI. By modelling every living entity in a game world, whether it be an animal such as a cow, a merchant trying to sell a player his wares or an enemy boss intent on destroying the hero, as an agent backed with an intelligent thought process, we can hope to bring the level of game AI to a new level. The result of which would be to create a better and more immersive experience for the player. Of course, we need more processing power to handle this compared to traditional game AI techniques. This is where the GPU comes in.

Video games aren't the only area that could benefit from combining agents with GPUs. Simulations such as pedestrian and traffic dynamics, ant hives, bacterial infections and many more can be modelled as agent based problems. The on-line market also stands to benefit from improvements in agent productivity[11].

That's not to say this idea isn't with out its issues. One does not simply move their code to a GPU and expect everything to work perfectly. In fact, in our particular case there are a lot of problems that need to be resolved in order to get agents to live happily on GPUs. After all, agents are meant to communicate with one another. How do you do this on a GPU? Agents also make decisions which implies divergence in our code. How can a GPU efficiently handle this? How can we minimise the overhead of moving data to and from the CPU? How do we keep our agents synchronised? How do we deal with slow access to global memory on the GPU?

2.1 Problems

2.1.1 How do we get Agents to communicate?

Message passing architectures already exist for both agents (KIF, KQML)[10] and parallel processing (MPI). So do we use one of these to handle communication between our agents? Well, not quite. These architectures are designed for distributed systems and as such would incur quite a bit of overhead in a non-distributed system such as the one we're working in. What we need to do is to make use of the GPUs global memory in the same way that a traditional multi-threaded CPU based program would make use of its shared memory. In the GPU case, we write messages to the GPU global memory and then have the relevant agent read that message and act accordingly. Of course, this leads us to another issue. How do we synchronise the reads and writes so as to not cause data corruption issues?

2.1.2 Data synchronisation

Traditionally you would use locking systems such as semaphores and mutexes to deal with data synchronisation in a CPU-based shared memory system. We, however, can exploit the design of the GPU and the GPGPU programming architectures which run on them to perform our synchronisation. The trick is to split our program into two separate kernels. In the first kernel we perform all the actions that write data to memory and the second kernel contains all the actions that perform reads from memory. We then queue the second action to occur after the first action has completed.

One other thing to keep in mind with regards to data synchronisation, is that most GPGPU architectures support barrier statements of some sort, allowing kernels to synchronise at a point in execution. However, these are generally specific to the set of SPs on an MP, not allowing for program wide synchronisation.

2.1.3 Minimising Overhead of moving data to and from the CPU

One of the big issues with developing for GPGPU is the overhead associated with moving data to the GPU and back to the CPU after processing is completed. One common way of dealing with this is to perform multiple runs of data processing on the GPU before moving it back to the CPU. We cannot do this in our case, however, as we want to present real-time output to the user of the application. To solve this issue, instead of processing the data a few times on the GPU and then brining it back to the CPU when we need to draw the results, why don't we just leave it on the GPU the whole time and draw the data directly from the GPU memory? In fact, it would be quite wasteful to process the data on the GPU and then send it to the CPU, which would send it back to the GPU for output. We can achieve this through interoperation.

2.1.4 How do we deal with high latency global memory access?

To keep hardware cheap and accessible, the largest part of the memory on a GPU is the global memory, which has relatively slow access times. Since this is a hardware issue, there isn't anything we can do to directly fix this problem. That said, we can attempt to hide some of the memory latency. In order to do this,

we need to keep the processing units busy with additional work. This can be achieved by having more threads than we have processors on the GPU. While one thread waits to access some memory, the MP will schedule a different thread to begin executing.

For our particular case where we are using agents instead of threads, we simply have to ensure that there exist more agents than SPs on the GPU. It's important to note that doing this doesn't give us better performance; it simply gives us better utilisation of our resources.

2.1.5 Handling Branch Divergence

Branch divergence has in the past been arguably the biggest deterrent from performing agent based simulations on the GPU. Agents need to make decisions. The implication of this being that the code will branch. How does a GPU handle this? How do we minimise the impact of this?

Simply put, if any of the threads running on one of the SPs of a MP diverge in execution from the other threads on the MP, all threads must execute the diverged segment of code[1]. The result being a massive waste of processing time for each of the threads on the MP that don't utilise that particular segment of code.

Han and Abdelrahman[4] have proposed two methods to help combat this issue: The first solution is Iteration Delaying, a runtime optimisation. Instead of letting threads execute whenever they like, threads that intend to follow the same path of execution are grouped up and executed together. Therefore, only that particular branch needs to be executed and no processing time is wasted. Obviously this approach does create quite a bit of overhead with sorting out which threads should execute when, but it can still result in a decrease in overall processing time. Han and Abdelrahman observed about a 30% reduction in processing time. The second proposed solution is Branch Distribution, which is a code optimisation. Branch Distribution works by extracting similar code sections from different branches of code and unifying this code to reduce the overall processing time. For example. Before Branch Distribution one code segment may read as such:

```
if (temp == true)
{
a = b + c;
}
else
{
a = d + e;
}
```

After Branch Distribution:

```
int x, y;
if (temp == true)
{
x = b;
y = c;
}
else
{
x = d;
y = e;
}
a = x + y;
```

With this change, rather than having two separate '+' operations, we only have it once. In the case where a GPU could be executing '+' twice, we have reduced it to only calculate it once. Of course, the cost is a higher memory footprint. However, even with this additional overhead, Han and Abdelrahman reported an 80% decrease in processing time.

The other thing to remember is that even with significant wasted processing time, we can still hope to see an increase in performance compared to a single core processor simply due to the brute force power of a GPU. For example, even if we waste as much as 75% of our processing time with a 100 agents, we're still looking at a (0.25 * 100) 25 times increase in speed. Obviously we've simplified a bit here. A CPU core runs faster than a SP as well as having a more advanced memory caching system.

3 Implementation

The system has been implemented in C/C++ using OpenCL to access GPGPU. OpenCL is an open standard which has been gaining popularity in the last few years and is backed and maintained by the Khronos consortium. It can be used for both AMD and Nvidia graphics cards and is programmed through a subset of C99 with some of its own extensions. These points make it a good choice for us to use in our test bed application.

OpenCL provides us with the ability to transfer data between itself and OpenGL through interoperation. This allows us to display data directly from GPU memory without the overhead of transferring data between the CPU and the GPU. While the current OpenCL specifications don't require interoperation[2], the current implementations of OpenCL from both AMD and Nvidia support this feature. Given how useful this feature is, we can expect that it will continue to receive support in the foreseeable future.

The system itself needs to implement a few important features to be considered a valid multi-agent system. Firstly, the agents need to be self-determining. That is, once they have been have launched, they need to be able to think independently of one another and be able to develop their own solutions to given problems without external assistance. Secondly they need to be able to carry out and execute their devised plans to the afore mentioned problems. Thirdly, and lastly, agents need to be able to communicate and bargain with one another in order to achieve their goals. With all of this in mind a simple test bed has been designed and developed. The remainder of this section will give an overview of the system and the details about how the proposed solutions to the problems listed in the previous section were implemented.

In our test bed system, the agents live in their own world with their primary goal being to stay alive. To stay healthy, agents must perform tasks such as drinking, eating and sleeping. In order to acquire water and food, they must purchase them using gold coins. To get gold coins, agents must perform some work. There are four jobs in this world: food farming, ice harvesting, food and water processing and shopkeeping. Farmers and harvesters sell their produce to processors for gold. The processors process the goods and sell them to shopkeepers. The shopkeepers then sell the processed food and water back to everyone. Buying and selling goods requires both communication and bargaining from the agents involved. A shopkeeper with a food shortage for example, won't be willing to part with his goods unless paid top dollar. For an agent to go to sleep, it simply has to go back to its designated bed.

Agent data is stored in the memory in the form of arrays which track the different properties of each agent, including details such as the position of the agent in a three dimensional world, how many money the agent has and whether or not the agent is tired, hungry or thirsty.

Agent communication has been implemented through global memory. Each agent has a segment of memory allocated to it for communication purposes. When one agent wants to communicate with another it must first check the memory of its potential communication partner and ensure that the target is not already communicating with another agent. If the target agent is busy (already has data in the memory), the agent trying to communicate will wait for the next loop of the simulation to try communication again. If the target agent is not busy, the communicating agent will write data into the memory of the target. The target agent will then check this memory on the next loop of the simulation, reading the actual message itself as well as the details about the agent who sent the message. The agent will then respond accordingly by writing data into the memory of the agent who opened the communications. This process will continue until the negotiations have ended.

Two implementations of the agent system have been developed using the above architecture. The first version runs on a CPU using C++ code. In this version agents are processed sequentially, one after the other. The second version runs on the GPU using OpenCL. In this version all the agent data is stored on the GPU memory. In addition, each agent is processed in parallel, with each agent being assigned its own processor during each loop of the application. Both versions of the application use OpenGL for graphical display and in both cases, agents perform deliberation during every loop. In addition, both versions of the code also use the same code base. That is, the GPU version of the code has been directly ported from the CPU version of the code with no GPU specific optimisations having been performed as of yet.

4 Results and Discussion

The system was tested on different hardware and with different numbers of agents. The CPU hardware used was a Intel Core 2 Duo T6500 2.10Ghz. The GPU hardware used was a Nvidia Geforce GT 220M. The system was tested with 1000 agents, 5,000 agents, 10,000 agents and 25,000 agents. The system was run in each configuration for a minute. The average execution time for each loop of the simulation was then calculated in milliseconds. The data has been graphed below. One thing to note is that the execution time includes both the processing time of the agent system as well as the time taken to draw the agents to the display. As discussed in section two, one of the main advantages of performing agent calculations on the GPU is the fact that we can minimise data transfers, reducing the amount of processing time overall. Therefore, we must ensure that this is taken into account when we measure the performance of the system.

The first graph, figure 2, shows the execution time for both pieces of hardware used against the amount of agents used in each particular test case.

The results displayed above do not paint the prettiest picture. In every case the CPU significantly outperformed the GPU. Even the worse case for the CPU with 25,000 agents ran in about half the execution time of the GPUs best case with 1,000 agents. One thing we can see however, is that the GPU appears to scale better than the CPU. By extrapolating the data in the chart above, we can generate a second graph, figure 3, which shows the scalability of both CPU and

ISBN: 978-1-61804-092-3

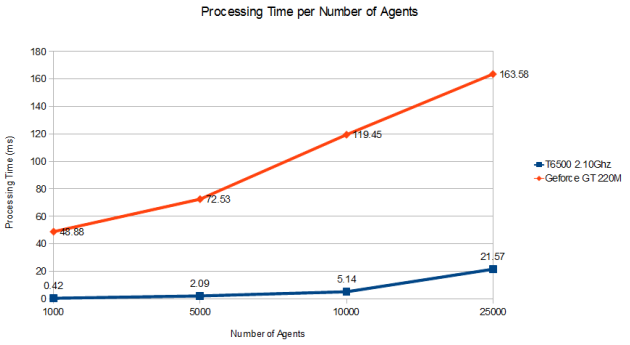
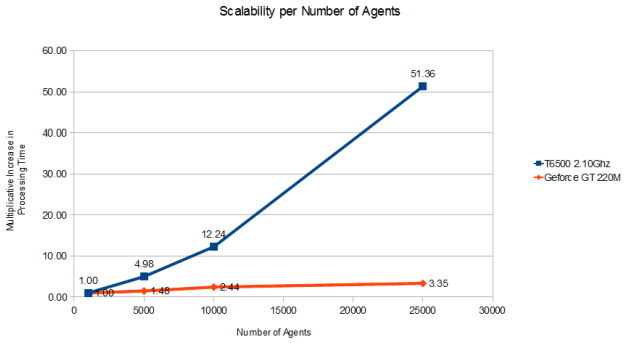


Figure 2: Execution Time

the GPU, with the comparison made against the 1,000 agents case.

Figure 3: Scalability



This graph gives us some hope, showing that as the number of agents increases, the percentage of extra processing time required grows at a significantly slower rate for the GPU version compared to the CPU version. For example, from 1,000 agents to 5,000 agents the GPU version only required 48% more processing time to handle the increase in agents while the CPU version required a 498% increase in processing time for the same step up in agents. The implication of this data is the GPU version has additional initial overhead, but scales at a much better rate compared to the CPU version which doesn't have the initial overhead but also doesn't scale as well. It could be expected that the processing time of the two different implementations will converge over time with the GPU version eventually surpassing the CPU version for exceedingly high numbers of agents.

We should also stop to consider a couple of other details. First of all, as mentioned at the end of the previous section, the code being used in the GPU version has not yet been optimised for the GPU. Specifically, the code still retains all the branches and loops as optimized for the CPU version, causing the code to perform many extra calculations and waste processor resources. Secondly, the system has only been tested on two pieces of hardware, both of which have relatively poor performance compared with current hardware in the fast moving computer hardware industry. Only within the last couple of years has GPU processing power really begun to outstrip CPU processing power. Tests performed on different hardware could certainly yield very different results.

5 Conclusions

While the GPU version of the software didn't provide the results that were hoped for, what it was able to show us was some solid scalability allowing us to the see the potential of a massively parallel multi-agent system on a GPU. With a combination of modern hardware, perfectly optimised GPU code and a large number of agents we could very well expect to see the brute force processing of a GPU outperform the intelligent processing of a CPU. As we continue to look towards the future, where the raw processing power of GPUs continues to grow at a phenomenal rate, we should expect to see more and more code finding its way across to the GPU.

6 Future Work

In section three we talked about problems encountered with executing agents on a GPU and we provided solutions to each of these problems. However, in our test bed system we have not yet implemented Branch Distribution and other code level optimisations and so the resulting test data is not the best that we could hope to see. With these optimisations completed we could certainly expect a large performance increase.

Secondly, our simulation was tested on a limited amount of hardware with only basic testing applied. With more hardware and a more thorough testing schedule we could not only expect to see different results, but we could also work to identify bottlenecks in the code, improving the overall performance.

Lastly, one thing that is quite important to games and some simulations is user input. Our particular test bed case does not handle user input at all. Reading input from a keyboard, getting it to the CPU and then moving that to GPU presents it's own set of challenges and problems that would be solved before we could truly see this type of parallel agent system used in a game.

References:

[1] B. R. Gaster, L. Howes, and D. Richie. Amd opencl webinar series. `http://developer.`

ISBN: 978-1-61804-092-3

amd.com/zones/OpenCLZone/Events/pages/OpenCLWebinars.aspx.

[2] K. Group. Opencl 1.2 specifications. www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[3] J. Hagelbäck and S. J. Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2*, AAMAS '08, pages 631–638, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[4] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[5] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1403–1410, New York, NY, USA, 2009. ACM.

[6] E. Manisterski, R. Lin, and S. Kraus. Understanding how people design trading agents over time. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3*, AAMAS '08, pages 1593–1596, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[7] T. Mine, K. Kimura, S. Amamiya, K. Takahashi, and M. Amamiya. Agent-community-network-based business matching and collaboration support system. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 75–78, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[8] D. W. Roeh, V. V. Kindratenko, and R. J. Brunner. Accelerating cosmological data analysis with graphics processors. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 1–8, New York, NY, USA, 2009. ACM.

[9] M. C. Toyama, A. L. C. Bazzan, and R. da Silva. An agent-based simulation of pedestrian dynamics: from lane formation to auditorium evacuation. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 108–110, New York, NY, USA, 2006. ACM.

[10] M. Woolridge and M. J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[11] G. Yamamoto and H. Tai. Performance evaluation of an agent server capable of hosting large numbers of agents. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS '01, pages 363–369, New York, NY, USA, 2001. ACM.