

An Adaptative Game Loop Architecture with Automatic Distribution of Tasks between CPU and GPU

Mark Joselli
UFF,Medialab

Marcelo Zamith
UFF,Medialab

Esteban Clua
UFF,Medialab

Anselmo Montenegro
UFF,Medialab

Regina Leal-Toledo
UFF,IC

Aura Conci
UFF,Medialab

Paulo Pagliosa
UFMS,DCT

Luis Valente
PUC-Rio,VisionLab

Bruno Feijó
PUC-Rio,VisionLab

Abstract

This paper presents a new architecture to implement any game loop models for games and real-time applications that uses the GPU as a Mathematics and Physics co-processor, working in parallel processing mode with the CPU. The model applies concepts of automatic task distribution. The architecture can apply a set of heuristics defined in Lua scripts to get acquainted about what is the best processor for handling a given task. The model applies the GPGPU (General-Purpose Computation on GPUs) paradigm. The architecture that this work proposes acquires knowledge about the hardware by running tasks in each processor, and by studying their performance over time, learning about what is the best processor for a group of tasks.

Keywords:: Game loops, GPGPU, Task distribution

Author's Contact:

{mjoselli, mzamith, esteban, anselmo, leal,aconci}@ic.uff.br
pagliosa@dct.ufms.br
{lvalente, bruno}@inf.puc-rio.br

1 Introduction

Real time systems, like games, are defined as solutions that have time constraints to run their tasks. So, if for any reason the system is not able to execute its work under some time threshold, it will fail. In order to achieve such constraints game loops can be implemented.

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Network and others. More, computer games are also interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition in games is related to gathering data from input devices as keyboards, mice and joysticks. Data processing tasks consist on applying game rules, responding to user commands, simulating Physics and Artificial Intelligence behaviors. Data presentation tasks relate to providing feedback to the player about the current game state, usually through images and audio.

Many computer games offer experiences where many actions seem to happen at once. However, computers usually have limited resources, so it is necessary to harvest the results of all processes involved in a game and present them to the player. If the application is not able to perform this work on time, the user may not receive continuous feedback, and the interactivity that the game should provide will not be acceptable. Hence, one of the main requirements of a game will not be fulfilled. This issue characterizes computer games as a heavy real-time application.

A common parameter to measure computer game performance is the number of frames per second (FPS) displayed on the screen. A frame represents an image displayed on the screen. A common accepted lower bound for interactive rates is 16 frames per second. Usually, a frame rate from 50 to 60 FPS is considered optimal.

Nowadays, computers and new video game consoles (such as the Xbox 360 and the Playstation 3) feature multicore processors. For this reason, game loops that take advantage of these resources are likely to become important in the near future. Therefore, parallelizing game tasks with multiple threads is a natural step. In order

to take advantage of different hardware, a generic architecture for game loops and a multi thread game loop with this architecture are present.

The development of programmable GPUs (graphics processing units), has enabled new possibilities for general purpose computation (GPGPU) which now can be used to process some of the common tasks of the game loop, like data processing tasks. This is good news for games due to the parallel architecture of the latest GPUs, which have more processing power than CPUs. GPUs perform better than CPUs when large amounts of data are involved, but to take advantage of this power, it necessary to develop a different approach than the traditional CPU sequential model. Hence, due to architectural characteristics, the CPUs are more suitable for processing small amounts of data while the GPUs are more suitable for large amounts of data. In order to achieve better performance in both cases (small and large amount of data), it is necessary to implement an automatic method to distribute tasks between the CPU and the GPU. For this heuristics to work better with different hardware architectures, they are implemented in a script language.

The main objective of this work is present a new game loop architecture that can be used to implement any game loop model and can take advantage of automatic dynamic distribution of tasks between the CPU and the GPU. This distribution is based on heuristics that are defined in Lua [Lua] scripts. This work presents concepts that could be applied also to other hardwares like the PlayStation 3 with the Cell Processor [Hofstee 2005].

This paper is organized as follows. Section 2 presents GPGPU concepts. Section 3 presents related works in game loops and task distribution between CPU and GPU. Section 4 presents the generic architecture for game loops. Section 5 presents the test case with the test game loop and the test heuristics, and Section 6 presents the results. Finally, Section 7 presents the conclusions.

2 GPGPU

Graphics Processors Units or simply GPUs are processors dedicated to mathematical processing in the graphics pipeline. The evolution of those processors allows it to be used for processing other mathematical tasks.

The GPUs have been evolving constantly and in a faster way than the CPUs, acquiring superior computation power. A nVidia 8800 ultra [NVIDIA 2006], for instance, can sustain a measured 384 GFLOPS's against 35.3 GFLOPS's for the 2.6 GHz dual core Intel Xeon 5150 [NVIDIA 2008]. This fact is attributed to the parallel SIMD architecture of the GPUs (the nVidia GeForce 9800 GX2, for example, has 256 unified stream processors). Because of the GPUs' parallel architecture, they are very good for processing applications that require high arithmetic rates and data bandwidths.

Nvidia and AMD/ATI are implementing unified architectures in their GPUs. Each architecture is associated with it a specific language: Nvidia has developed CUDA (Compute Unified Architecture) [nVidia 2007b] and AMD developed CAL (Compute Abstraction Layer) [AMD 2007]. One main advantage in the use of these languages is that they allow the use of the GPU in a more flexible way (both languages are based on the C language) without some of the traditional shader languages limitations such as "scatter" memory operations, i.e. indexed write array operations, and others that are not even implemented as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts [Owens et al. 2007]. Nevertheless, the disadvantage of these architectures is that

they are only available for the vendors of the software, i.e., CUDA only works on Nvidia and CAL only works on AMD/ATI cards. In order to have GPGPU programs that work on both GPUs it is necessary to implement them in shader languages like GLSL (OpenGL Shading Language), HLSL (High Level Shader Language) or CG (C for Graphics) with all the vertex and pixel shader limitations and idiosyncrasies.

In addition, Intel has recently presented a new architecture for GPUs called Larrabee [Seiler et al. 2008]. It is made up of several x86 processors in parallel which can be used to process both graphics and non-graphics data. The advantage of this architecture is that it does not need a special language, just plain C. Nevertheless, it will only be available in late 2009.

There are many areas that apply GPGPU: wheather forecast [Michalakos and Vachharajani 2008], chemistry [Ufimtsev and Martnez 2008] and of course graphics. Games use GPGPU mainly in two areas: Physics and AI.

PhysX [Ageia 2008] and Havok [Intel 2008] are examples of physics engines which have used the GPU to accelerate their physics loop (eight times of speedup in the case of Havok [Green 2007]). Also the book GPU Gems 3 [Nguyen 2007] presents a full section dedicated for physics using the GPGPU. In the field of AI it can be seen implementations of state machines [Rudomn et al. 2005] and flocking boids [Erra et al. 2004] using the GPU to processes its data.

3 Related works

The game loop can be divided in three general classes of tasks:

- Data acquisition task is responsible for getting user commands from the various input devices;
- Data processing tasks, also referred as the update stage, are responsible for tasks that update the game state, for example: character animation, Physics simulation, Artificial Intelligence, game logic, and network data acquisition;
- Data presentation task is responsible for presenting the results to the user. In games, this corresponds usually to rendering graphics and playing audio.

The main objective of real-time game loop models in the literature is to arrange the execution of those tasks, in order to simulate parallelism. The work by Valente et al [Valente et al. 2005] provides a survey of real-time game loop models hat regards single-player computer games. But it does not cover the use of GPGPU as an update stage of the game loop.

The simplest implementation of a game loop with the GPGPU as an update stage is executing it sequentially, as shown in figure 1. Diverse GPGPU implementations use this game loop, as the CUDA particles demo [nVidia 2007a].

Another game loop with GPGPU presented in the literature is the multithread architecture with GPGPU stage uncoupled from the main loop [Joselli et al. 2008a; Joselli et al. 2008b]. This architecture is composed by two threads. One of them gathers user input, executes rendering, and updates the game state. The other thread runs the GPGPU. Figure 2 illustrates this game loop model.

The multithread uncoupled with a GPGPU stage [Zamith et al. 2007] is the other game loop with GPGPU available on the literature. This game loop consists of three threads: the first deals with gathering user input and updating the game state. The second thread is responsible for rendering the scene. The third one runs the GPGPU. Figure 3 illustrates this game loop.

The literature on task distribution between CPU and GPU is scarce. The work by [Zamith et al. 2007] implements a semi-automatic task scheduling distribution between CPU and GPU via a script file. Joselli et al. [Joselli et al. 2008a; Joselli et al. 2008b] implements some heuristics for automatic task distribution between CPU and GPU, using a Physics engine that has some methods implemented in both CPU and GPU.

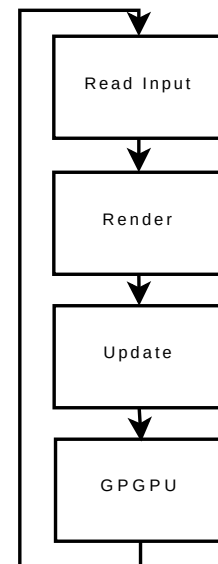


Figure 1: Single coupled loop

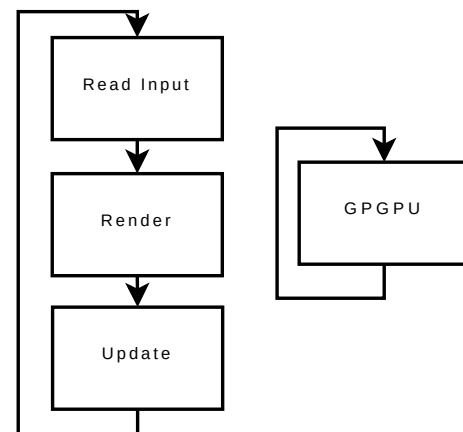


Figure 2: Multithread architecture with GPGPU stage uncoupled from the main loop

All of those game loops can be implemented in the adaptive game loop architecture presented in the next section.

4 The Adaptive Game Loop Architecture

This paper presents a new game loop architecture, named Adaptive Game Loop Architecture. This architecture is able to:

- Implement the game loop by using a multi-threads or a single-thread mode;
- Use coupled and uncoupled tasks;
- Use pixel shader or CUDA.

This architecture is based on the concept of tasks. A task corresponds to some work that the application should execute, for example: reading player input, rendering and update application objects.

In the proposed architecture, a task can be anything that the application should work towards processing. However, not all tasks can be processed by all processors. Therefore, the application has three groups of tasks. The first one consists of tasks that can be modeled only for running on the CPU, like reading player input, file handling, and managing other tasks. The second group consists of tasks that can only run in the GPU, like the presentation of the scene. The third group can also be modeled for running on both processors. These tasks are responsible for updating the state of some objects that belongs to the application, like AI and Physics.

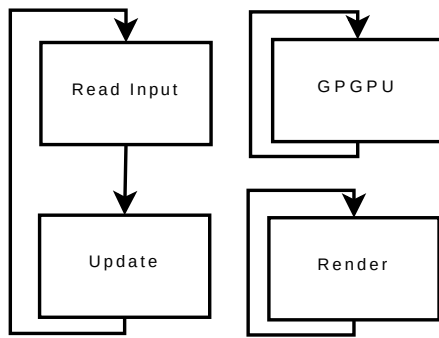


Figure 3: Multithread uncoupled with an GPGPU stage

The task concept is modeled as an abstract class that different threads are able to load. Figure 4 illustrates the UML class diagram for the Task and its subclasses.

The Task class is the virtual base class and has four subclasses: Input Task, Update Task, Presentation Task, and Automatic Update Task. The first three are also abstract classes. The latter is a special class whose work is to perform the automatic dynamic distribution between the CPU and the GPU. This distribution consists of choosing the processor that is going to run a task according to some heuristic specified in a script file. Also a special class, the Task Manager class, is responsible for creating and keeping all the tasks of the game loop (discussed in Subsection 4.1).

The Input Task classes and subclasses handle user input related issues. The Update Task classes and subclasses are responsible for updating the game state. The CPU Update class should be used for tasks that run on the CPU, and the GPU Update class corresponds to tasks that run on the GPU. The Presentation Task and subclasses are responsible for presenting information to the user, which can be visual (Render Task) or sonic (Sound Task).

4.1 The Task Manager

The Task Manager (TM) is the core component of the proposed architecture. It is responsible for instantiating, managing, synchronizing, and finalizing task threads. Each thread is responsible for tasks that run either on the CPU or on the GPU. In order to configure the execution of the tasks, each task has control variables described as follows:

- **THREADID**: an id of the thread that the task is going to use. When the TM creates a new thread, it creates a **THREADID** for the thread and it assigns the same id to every task that executes in that thread;
- **UNIQUEID**: an unique id of the task. It is used to identify the tasks;
- **TASKTYPE**: the task type. The following types are available: input, update, presentation, and manage;
- **DEPENDENCY**: a list of the tasks (ids) that this task depends on to execute.

With that information, the TM creates the task and configures how the task is going to execute. A task manager can also hold another task manager, so to use it to manage some distinct group of tasks. An example of this case is the automatic update tasks that Subsection 4.2 presents.

The Task Manager acts as a server and the tasks act as its clients, as every time a task ends, it sends a message to the Task Manager. The Task manager then checks which task it should execute in the thread.

When the Task Manager uses a multi-thread game loop, it is necessary to apply a parallel programming model to be able to identify the shared and non-shared sections of the application, because they should be treated differently. The independent sections compose tasks that are processed in parallel, like the rendering task. The

shared sections, like the update tasks, need to be synchronized in to guarantee mutual-exclusive access to shared data and to preserve task execution ordering.

Although the threads run independently from each other, it is necessary to ensure the execution order of some tasks that have processing dependence. The architecture accomplishes this by using the **DEPENDENCY** variable list that the Task Manager checks to know the task execution ordering.

The processing dependence of shared objects needs to use a synchronization object, as applications that use many threads do. Multi thread programming is a complex subject, because the tasks in the application run alternately or simultaneously, but not linearly. Hence, synchronization objects are tools for handling task dependence and execution ordering. This measure should also be carefully applied in order to avoid thread starvation and deadlocks. The TM uses semaphores as the synchronization object.

4.2 The Automatic Update Task

The purpose of this class is to define which processor will run the task. The class may change the task's processor during the application execution, which characterizes a dynamic distribution.

One of the major features of this new architecture is to allow dynamic and automatic task allocation between the CPU and GPU, in order to do that it uses the Automatic Update Task class. This task can be configured in order to be executed in three modes: CPU only, GPU only and in the automatic distribution between CPU and GPU. In order to execute on the CPU a CPU implementation must be provided, and in the GPU a GPU implementation must be provided, and in order to make use of the automatic distribution both implementation must be provided. The scheduling is by heuristic in a script file. Also a configuration on how the heuristic is going to behave is needed, and for that a script configuration file is presented in Subsection 4.2.1. The scripts files are implemented in Lua [Ierusalimsky et al. 2006] (Subsection 4.2.2).

The Automatic Update Task acts like a server and its tasks as clients. The role of the automatic update task is to execute a heuristic to automatic determine in which processor the task will be executed. The Automatic update task executes the heuristic and determines which client will execute the next task and will send a message to the chosen client, allowing it to execute. Also, every time the clients finish a task they send a message to the server to let it know it has finished. Figure 5 illustrate this process.

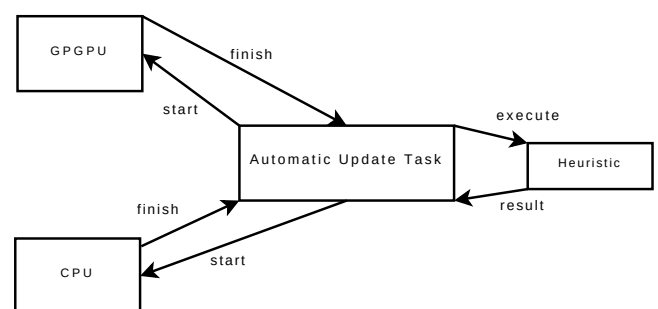


Figure 5: The Automatic Update Task class and messages

4.2.1 The Configuration Script

The configuration script is used in order to configure how the automatic update task will execute the heuristic. This script defines four variables:

- **INITFRAMES**: used in order to set how many frames are used by the heuristic to do the initial tests. These initial tests are used in order that the user may want that the heuristic make the initial tests different than the normal test;
- **DISCARDFRAME**: used in order to discards the first **DISCARDFRAME** frame results, because the main thread can be loading images or models and it can affect the tests;

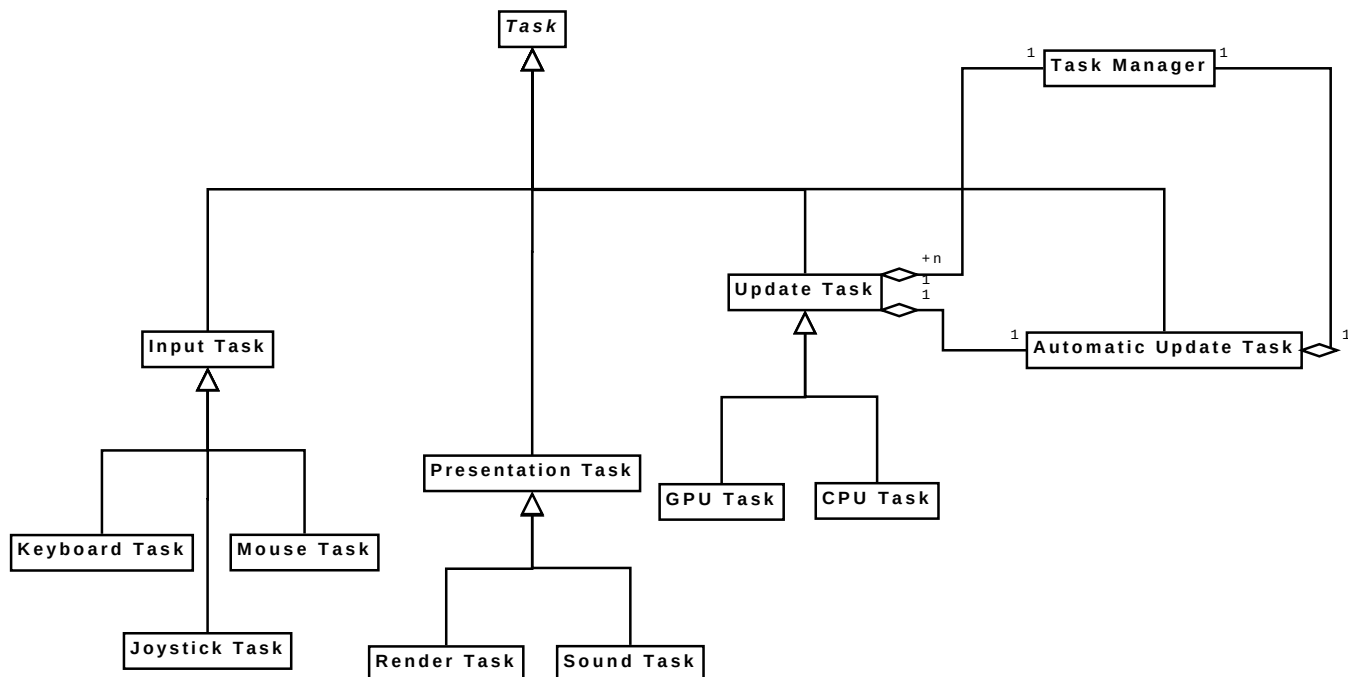


Figure 4: Multithread uncoupled with an GPGPU stage

- **LOOPFRAMES**: it is used to set on how frequency the heuristic will be executed. If this value is set to -1 the heuristic will be executed only once;
- **EXECUTEFRAMES**: it is used to set how many frames are needed before the decision on changing the processor will execute the next tasks.

An example of the configuration scrip file can be seen in script 1.

Script 1 Configuration Script

```
INITFRAMES    = 20
DISCARDFRAME  = 5
LOOPFRAMES    = 50
EXECUTEFRAMES = 5
```

So the automatic update task begins executing after the DISCARDFRAME are executed, then it execute INITFRAMES frames in the CPU and the next INITFRAMES in the GPU then it decides where the next LOOPFRAMES frames will be executed. If the LOOPFRAMES is greater then -1, it executes EXECUTEFRAMES frames in the CPU and it executes EXECUTEFRAMES frames in the GPU then it decides where the next LOOPFRAMES frames will be executed and keep repeating until the application is aborted.

4.2.2 The Heuristic Script

The heuristic script is used in order to distribute automatically the tasks between the CPU and the GPU. This script defines three functions:

- **reset()**: reset all the variables that the script uses in order to decide which processor will execute the task. This function is called after the LOOPFRAMES frames are executed. The variable that are normally used by the heuristic are:
 - **CPUTime**: this is the sum of all the elapsed times that the task has been processed in the CPU;
 - **GPUPTime**: this is the sum of all the elapsed times that the task has been processed in the GPU;
 - **bestCPUFPS**: the best frame rate achieved by the CPU;
 - **bestGPUFPS**: the best frame rate achieved by the GPU;
- **setVariable(elapsedTime, processor)**: in this function is where all the variables that are used by the heuristic are set. This

function is called after the EXECUTEFRAMES frames in each processor. This function can be seen on script 2.

Script 2 setVariable Script

```
function setVariable(elapsedTime, processor)
  FPS = 1 / elapsedTime
  if (processor == CPU) then
    CPUTime = CPUTime + elapsedTime
    if (FPS > bestCPUFPS) then
      bestCPUFPS = FPS
    end
  else
    GPUPTime = GPUPTime + elapsedTime
    if (FPS > bestGPUFPS) then
      bestGPUFPS = FPS
    end
  end
end
```

- **main ()**: This is the function that executes the heuristic and decides which processor will execute the task. This function is called just before the LOOPFRAMES frames are executed. A script with this function implemented with the decision of always executing in the GPU can be seen on script 3.

Script 3 Main Script

```
function main()
  return GPU;
end
```

5 Test Case

The test case corresponds to the n-bodies sample [Nyland et al. 2007] from the GPU Gems 3 [Nguyen 2007]. The authors had implemented this example only to validate the model of game loop proposed in this work, because this problem works with intense mathematics processing.

The n-bodies demo is an approximation of the evolution of a system of bodies in which every body interacts with every other body. It also applies to different simulations like protein folding, turbulent fluids, global illumination and astrophysics. In this case the n-bodies is a simulation of astrophysics in which each body repre-

sents a galaxy or an individual star, and each body attracts or repels each other with gravitational forces.

This sample was implemented in both CPU and GPU. The GPU version uses CUDA. It is important to remark that even though the demo uses CUDA, the game loop implementation could use CAL or shader languages (GLSL, HLSL or CG) without major modifications in the framework layer. Figure 6 illustrates a set of frames from the simulation.

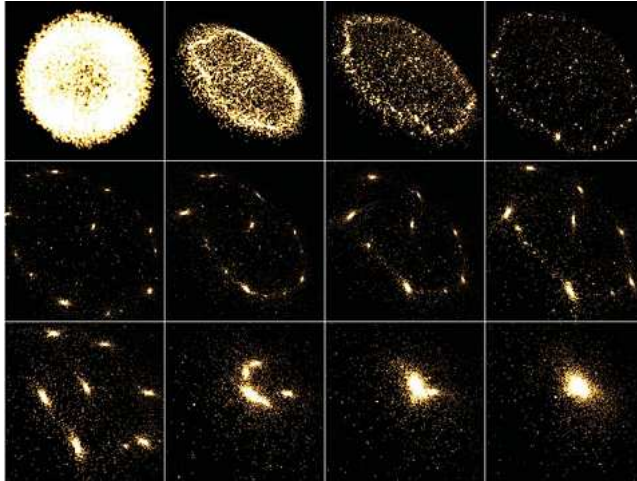


Figure 6: *N*-bodies sample

The authors did not emphasize the *n*-bodies problem, because it is not the aim of this work. The *n*-bodies is used such as example only for validate the method proposed.

5.1 The Tested Heuristic

The demo uses a very simple heuristic. It checks which processor was the fastest one in running the task and then selects this processor to run the next frames. Script 4 lists the heuristic.

Script 4 Tested Heuristic

```
function main()
  if (CPUtime < GPUtime) then
    mode = CPU;
  else
    mode = GPU;
  end
end
```

The heuristic is configured in script, and any kind of heuristics can be implemented, but the heuristics developed by the authors can work in two different ways:

- The first, that is called initial, is configured in order to execute 20 frames in the CPU and 20 frames in the GPU and then it decides for the fastest processor.
- The second, that is called looped, is configured to loop in the following state: execute 5 frames in the CPU and 5 frames in the GPU and decide to for the fastest processor to execute the next 200 frames.

5.2 The Tested Game Loop

To test the architecture, the demo implemented a game loop with an input task and a render task in the main loop, and an automatic update task with CPU/GPU in another thread (uncoupled). Figure 7 illustrates this game loop.

6 Results

The tests were done base on the fast *n*-bodies simulation with CUDA, such as described before. There are two groups of tests.

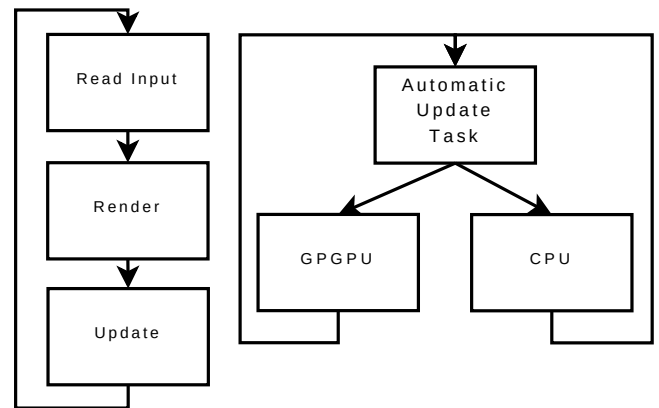


Figure 7: The Multi thread loop with an automatic update task uncoupled from the main loop.

The first group uses the initial heuristic where the fastest processor is selected at the beginning of the task execution. The second group uses the other heuristic (looped), that is, the heuristic is invoked for each cycle of the frames. The CPU tests were made with an Intel quad-core 2.4 GHz and the GPU tests were made with three different GPUs a nVidia Geforce 8800 GTS, a nVidia Geforce 8400 GS and a nVidia 8200M G.

For both groups, the example was executing the application and the work of heuristic to choice the processor. Table 1 illustrate the performance of the application and the processing for both processors. The initial number of bodies is 4 and it is increased until 8192 bodies. Figure 8 shows a comparison between the CPU and the nVidia 8800 GTS GPU when there are few number of bodies.

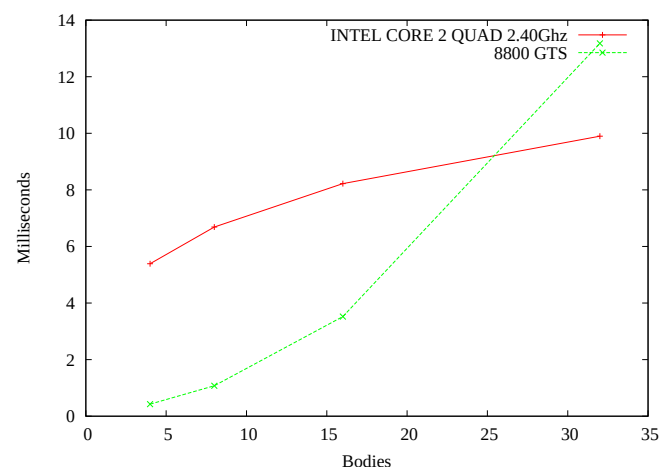


Figure 8: Comparison between the CPU and the GPU

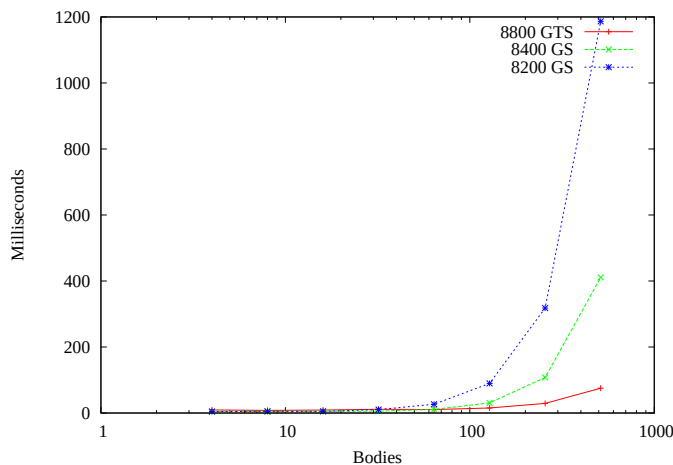
Although the GPU is faster than CPU, until 25 bodies, approximately, the CPU is better and the heuristic chooses it, after the GPU is already chosen. So, the CPU is faster with less bodies and the GPU, in this example, is more efficient with higher number of bodies. Figure 9 shows the comparison between the evolution of the tested GPUs.

7 Conclusion

Multicore hardware architectures are a tendency, and both the CPU and the GPU have developed great evolution in this aspect. Quad-code processors are present in the latest CPU architecture and unified architecture is presented in the latest GPUs. This tendency is not only in increasing the processing power but also increasing the number of cores available. With that the parallel processing, in this architectures, is a reality. With this hardware evolution the games will get much more sophisticated and multicore game loops with the use of the GPU will get more common.

Table 1: Elapsed time of processors in 100 iterations measured in milliseconds

bodies	CPU elapsed time	8800 GTS elapsed time	8400 GS elapsed time	8200M G elapsed time	processor
4	0.404	7.792	4.251	4.666	CPU
8	1.010	9.170	4.417	5.105	CPU
16	3.279	9.265	4.756	5.116	CPU
32	12.243	10.600	5.700	10.343	GPU
64	48.001	11.250	10.931	26.496	GPU
128	190.745	15.182	30.658	89.628	GPU
256	773.152	29.244	107.664	318.036	GPU
512	3124.517	75.188	410.865	1186.663	GPU
1024	12155.210	282.648	1619.403	4584.704	GPU
2048	48627.184	989.581	6526.119	18097.682	GPU
4096	195216.563	3835.552	25815.580	71998.977	GPU

**Figure 9:** Comparison between the GPUs

References

- AGEIA, 2008. Physx. Available at: <http://www.ageia.com>. 20/02/2008.
- AMD, 2007. Amd stream computing. Available at: <http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf>. 20/02/2008.
- ERRA, U., CHIARA, R. D., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. *Vision, Modeling, and Visualization*, 233–240.
- GREEN, S., 2007. Gpgpu physics. Siggraph07 GPGPU Tutorial.
- HOFSTEE, H. P. 2005. Power efficient processor architecture and the cell processor. *IEEE Proceedings of the 11th International Symposium on High-Performance Architecture*.
- IERUSALIMSKY, R., DE FIGUEIREDO, L. H., AND CELES, W. 2006. *Lua 5.1 Reference Manual*. Lua.org.
- INTEL, 2008. Havok. Available at: <http://www.havok.com>. 20/02/2008.
- JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., FEIJÓ, B., DORNELLAS, M., LEAL, R., AND POZZER, C. 2008. Automatic dynamic task distribution between cpu and gpu for real-time systems. *IEEE Proceedings of the 11th International Conference on Computational Science and Engineering*, 48–55.
- JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 149–156.
- LUA. The programming language lua. Disponível em: <http://www.lua.org/manual/>. 20/12/2007.
- MICHALAKES, J., AND VACHHARAJANI, M. 2008. Gpu acceleration of numerical weather prediction. *IEEE International Symposium on Parallel and Distributed Processing*, 1–7.
- NGUYEN, H. 2007. *GPU Gems 3 - Programming Techniques for High-performance Graphics and General-Purpose Computation*. Addison-Wesley.
- NVIDIA. 2006. Geforce 8800 gpu architecture overview. tb-02787-001_v0.9. Technical report, NVIDIA.
- NVIDIA, 2007. Cuda particles. Available at: <http://developer.download.nvidia.com/compute/cuda/1.1/website/projects/particles/doc/particles.pdf>. 20/02/2008.
- NVIDIA, 2007. Nvidia cuda compute unified device architecture documentation version 1.1. Available at: <http://developer.nvidia.com/object/cuda.html>. 20/12/2007.
- NVIDIA. 2008. Nvidia - cuda compute unified device architecture. Programming guide, NVIDIA.
- NYLAND, L., HARRIS, M., AND PRINS, J. 2007. Fast n-body simulation with cuda. *GPU Gems 3 Chapter 31*, 677–695.
- OWENS, J. D., LEUBKE, D., GOVINDARAJU, N., HARRIS, M., KRÄGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1), 80–113.
- RUDOMĂN, T., MILLĂN, E., AND HERNÁNDEZ, B. 2005. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory* 13(8), 741–751.
- SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics* 27, 3.
- UFIMTSEV, I. S., AND MARTÍNEZ, T. J. 2008. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal Chemistry Theory Computation* 4 (2), 222 – 231.
- VALENTE, L., CONCI, A., AND FEIJÓ, B. 2005. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 89–99.
- ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2007. The gpu used as a math co-processor in real time applications. *Proceedings of the VI Brazilian Symposium on Computer Games and Digital Entertainment*, 37–43.