The GPU Used as a Math Co-Processor in Real Time Applications

Marcelo P. M. Zamith
Esteban W. G. Clua
Aura Conci
Anselmo Montenegro
Instituto de Computação
Universidade Federal Fluminense

Paulo A. Pagliosa
Departamento de Computação e Estatística
Universidade Federal de Mato Grosso do Sul

Luis Valente
VisionLab/IGames
Departamento de Informática
PUC-Rio

Abstract

This paper presents the use of GPU as a math co-processor in real-time applications, in special games and physics simulation. Game loop architecture is used to validate the use of GPU such as math and physics co-processor, thus it is shown by this paper a new game loop architecture that employs graphics processors (GPUs) for general-purpose computation (GPGPU). A critical issue in this field is the process distribution between CPU and GPU. The presented architecture consists in a model for distribution and our implementation showed many advantages in comparison to other approaches without a GPGPU stage.

The architecture presented here was mainly designed to support mathematics and physics on the GPU, therefore the GPU stage in the model proposed works to help the CPU such as a math co-processor. Nevertheless, any kind of generic computation can be adapted. The model is implemented in an open source game engine and results obtained using this platform are presented.

Keywords:: game loop, GPGPU, co-processor.

Author's Contact:

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

1 Introduction

New graphics hardware architectures are being developed with technologies that allow more generic computation. GPGPU (general-purpose computation on GPUs) became very important and started a new area related to computer graphics research. This leads to a new paradigm, where the CPU — central processing unit — does not need to compute every non-graphics application issue. However, not every kind of algorithm can be allocated for the GPU — graphics processing unit — but only those that can be reduced to a stream based process. Besides, even if a problem is adequate for GPU processing, there can be cases where using the GPU to solve such problems is not worthy, because the latency generated by memory manipulation on the GPU can be too high, severely degrading application performance.

Many mathematics and physics simulation problems can be formulated as stream based processes, making it possible to distribute them naturally between the CPU and the GPU. This may be extremely useful when real time processing is required or when performance is critical. However, this approach is not always the most appropriate for a process that can be potentially solved using graphics hardware. There are many factors that must be considered before deciding if the process must allocate the CPU or the GPU. Some of these factors may be fixed and some may depend on the process status.

A correct process distribution management is important for two reasons:

- It is desired that both the GPU and the CPU have similar process load, avoiding the cases where one is overloaded and the other is fully idle;
- It is convenient to distribute processes considering which architecture will be more efficient for that kind of problem.

For instance, many real time graphics applications render the scene more than 60 frames per second, but execute physics or artificial intelligence simulations less frequently. Almost every video display has a refresh rate of 60 Hz, when rendering more than 60 frames per second, many of the calculated frames are discarded. In this case, it would be convenient to reduce graphics calculations and increase the other ones.

As it is not always possible to tell which processing route (GPU or CPU) a problem should go through, it is important that the framework or engine being used for the application development to be responsible for dynamically allocating jobs.

This paper proposes a new architecture for a correct and efficient semi-automatic process load distribution, considering real time applications like games or virtual reality environments. The use of GPU in this model has focus to help the CPU at processing of mathematics and physics such as mathematic co-processor. The presented idea is implemented in an academic open source game engine [Valente 2005].

The paper is organized as follows. Section 2 summarizes the main functionalities of a physics engine currently being developed and that is integrated into the academic framework, as a component. The purpose of that section is to identify which math operations are more suitable for implementation on GPU. Section 3 presents related works. Section 4 describes the architecture of framework adopted and its GPGPU shader support, as well as a first model for distributing tasks between CPU and GPU. Section 5 presents a simple example and results. Finally, Section 6 points out conclusions and future works.

2 Math and Physics on GPU

The ODE (Open Dynamics Engine) was integrated to the framework by the authors. This open source component is responsabile for real time dynamic simulation of rigid and elastic bodies and corresponds to one of the state-of-the-art tools available for physics simulation. A rigid body is a (possibly continuum) particle system in which the relative distance between any two particles never changes, despite the external forces acting on the system. The *state* $\mathbf{X}(t)$ of a body at time t is defined as [Feijó et al. 2006]:

$$\mathbf{X}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{q}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{bmatrix}, \tag{1}$$

where $\mathbf{X}$ is the world position of the center of mass of the body, $\mathbf{q}$ is a quaternion that represents the rotation of the local reference frame of the body in relation to the world frame, $\mathbf{P}$ is the world linear momentum and $\mathbf{L}$ is the world angular momentum of the body. The main functionality of a physics engine is, known the state $\mathbf{X}(t)$, to determine the next state $\mathbf{X}(t + \Delta t)$ of each rigid body into the scene, where Δt is the time step. This task involves the integration of the *equation of motion* of a rigid body [Feijó et al. 2006]:

$$\frac{d}{dt}\mathbf{X}(t) = \begin{bmatrix} \mathbf{v}(t) \\ \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t) \\ \mathbf{F}(t) \\ \boldsymbol{\tau}(t) \end{bmatrix}, \tag{2}$$

where $\mathbf{v} = m^{-1}\mathbf{P}$ is the world linear velocity of the center of mass of the body, $\mathbf{w}$ is the quaternion $[0, \boldsymbol{\omega}]$, $\boldsymbol{\omega} = \mathbf{I}^{-1}\mathbf{L}$ is the angular velocity of the body, $\mathbf{F}$ is the external force and $\boldsymbol{\tau}$ is the external

torque applied to the body, and m and $\mathbf{I}$ are the mass and *inertia tensor* of the body at time t, respectively.

Equation (2) is a first order ordinary differential equation (ODE); the component of a physics engine responsible by its integration (usually by applying a numerical method such as Runge-Kutta fourth-order) is called *ODE solver*.

Generally, the motion of a rigid body is not free, but subject to *constraints* that restraint one or more *degrees of freedom* (DOFs) of the body. Each constraint applied to a body introduces an unknown *constraint force* that should be determined by the physics engine in order to assure the restriction of the corresponding DOF. Constraints can be due to *joints* between (usually) two bodies and/or (*collision* or *resting*) *contact* between two or more bodies [Feijó et al. 2006].

In order to compute the contact forces that will prevent interpenetration of bodies, a physics engine needs to know at time t the set of *contact points* between each pair of bodies into the scene. The contact information includes the position and surface normal at the contact point, among others. This task is performed by a component integrated to the engine responsible for *collision detection*, which can be divided in a *broad* and a *narrow* phase [Ericson 2005]. In the broad phase only a sort of (hierarchies of) bounding volumes containing the more complex geometric shapes of the bodies are checked for intersecting; if they do not intersect, their bodies do not collide. Otherwise, in the narrow phase the shapes themselves are checked for intersecting and the contact information is computed.

Once found the contact points, the physics engine must simultaneously compute both the contact and joint forces and applies them to the respective bodies. Mathematically, this can be formulated as a *mixed linear complementary problem* (LCP), which can be solved, for example, by using the Lenke's algorithm [Eberly 2004]. The task is performed by a component of the engine called *LCP solver*.

In short, the main tasks performed at each time step during the simulation of a scene made of rigid bodies are: collision detection, collision handling, and resolution of differential equations.

For collision detection, GPUs can be used as a co-processor for accelerating mathematics or for fast image-space-based intersection techniques [Ericson 2005]. These ones rely on rasterizing the objects of a collision query into color, depth, or stencil buffers and from that performing either 2D or 2.5D overlap tests to determine whether the objects are in intersection [Baciu and Wong 2003; Govindaraju et al. 2003; Heidelberger et al. 2004]. Section 5 presents an illustrative implementation.

The ODE solver is a component that can be also efficiently implemented on GPU, since the integration of Equation (2) for a rigid body is performed independently of the other ones (therefore in parallel). Besides, data in Equations (1) and (2), which are related to the state of a rigid body and its derivative, can be easily write to and read from streams; the implementation of a method such as a Runge-Kutta solver on GPU (the arithmetic kernel) is also straightforward.

In the literature there are many works on solving ODEs on GPU, especially those related to (discrete) particle system simulation. For example, Kipfer et al. presented a method for this including inter-particle collisions by using the GPU to quickly sort the particles to determine potential colliding pairs [Kipfer et al. 2004]. In a simultaneous work, Kolb et al. produced a GPU particle system simulator that supported accurate collisions of particles with scene geometry by using GPU depth comparisons to detect penetration [Kolb et al. 2004]. Related to particle systems is cloth simulation, Green demonstrated a very simple cloth simulation using Verlet integration with basic orthogonal grid constraints [Green 2003]. Zeller extended this work with shear constraints that can be interactively broken by the user to simulate cutting of the cloth into multiple pieces [Zeller 2005].

Realistic physical simulation of deformable solid bodies is more complicated than rigid ones and involves the employment of numerical methods for solving the partial differential equations (PDEs) that govern the behavior of the bodies. Such methods are based on a subdivision of the volume or surface of a solid in a mesh of discrete elements (e.g. tetrahedrons or triangles); mathematically, the PDEs are transformed in systems of equations that, once solved, give the solution of the problem at vertices of the mesh.

Two domain techniques are the finite differences and finite element methods (FEM). The former has been much more common in GPU applications due to the natural mapping of regular grids to the texture sampling hardware of GPUs. Most of this work has focused on solving the pressure-Poisson equation that arises in the discrete form of the Navier-Stokes equations for incompressible fluid flow. The earliest work on using GPUs to solve PDEs was done by Rumpf and Strzodka [Rumpf and Strzodka 2005], where they discuss the use of finite element schemes for PDE solvers on GPUs in detail.

The current research of some of the authors on deformable bodies initially considers perfect linear solids only, which are governed by Navier equation. The solving technique is based on the boundary element method (BEM) with use of the Sherman-Morrison-Woodbury formula to achieve real time responses, as suggested in the work of James and Pai [James and Pai 1999]. One of the possibilities of GPGPU is to use the GPU as a math co-processor for implementation of a number of techniques for numerical computing, mainly those ones for matrix operations and solving linear systems of equations.

The design of an architecture for process distribution is a critical issue for an efficient collaboration between CPU and GPU. The present work implements some of the mentioned calculations in order to validate and test this distribution strategy. Very good results are achieved as discussed in Section 5.

3 Related Works

GPGPU is a research area in expansion and much promising early work has appeared in the literature. Owens et al. present a survey on GPGPU applications, which range from numeric computing operations, to non-traditional computer graphics processes, to physical simulations and game physics, and to data mining, among others [Owens et al. 2007]. This section cites works related to math and physics of solids on GPU.

Bolz et al. [Bolz et al. 2003] presented a representation for matrices and vectors on GPU. They implemented a sparse matrix conjugate gradient solver and a regular grid multigrid solver for GPUs, and demonstrated the effectiveness of their approach by using these solvers for mesh smoothing and solving the incompressible Navier-Stokes equations.

Krüger and Westermann took a broader approach and presented a general linear algebra framework supporting basic operations on GPU-optimized representations of vectors, dense matrices, and multiple types of sparse matrices [Krüger and Westermann 2003]. Using this set of operations, encapsulated into C++ classes, Krüger and Westermann enabled more complex algorithms to be built without knowledge of the underlying GPU implementation.

Galoppo et al. [Galoppo et al. 2005] presented an approach to efficiently solve dense linear systems. In contrast to the sparse matrix approaches, they stored the entire matrix as a single 2D texture, allowing them to efficiently modify matrix entries. The results show that even for dense matrices the GPU can to outperform highly optimized ATLAS implementations.

Fatahalian et al. did a general evaluation of the suitability of GPUs for linear algebra operations [Fatahalian et al. 2004]. They focused on matrix-matrix multiplication and discovered that these operations are strongly limited by memory bandwidth when implemented on the GPU. They explained the reasons for this behavior and proposed architectural changes to improve GPU linear algebra performance.

Full floating point support in GPUs has enabled the next step in physically based simulation: finite difference and finite element techniques for the solution of systems of partial differential equations. Spring-mass dynamics on a mesh were used to implement basic cloth simulation on a GPU [Green 2003; Zeller 2005].

Recently, NVIDIA and Havok have been shown that rigid body simulations for computer games perform very well on GPUs [Bond 2006]. They demonstrated an API, called Havok FX, for rigid body and particle simulation on GPUs, featuring full collisions between rigid bodies and particles, as well as support for simulating and rendering on separate GPUs in a multi-GPU system. Running on a PC with dual NVIDIA GeForce 7900 GTX GPUs and a dual-core AMD Athlon 64 X2 CPU, Havok FX achieves more than a 10 times speedup running on GPUs compared to an equivalent, highly optimized multithreaded CPU implementation running on the dual-core CPU alone.

Havok FX supports a new type of rigid body object called debris primitive. This one is a compact representation of a 3D object on possible collision that can be processed via shader model 3.0 (SM3.0) in a very efficient manner. Debris primitives can be premodeled as part of a game's static art content or generated on the fly during game play by the CPU, based on the direction and intensity of a force (e.g. brick and stone structure blown apart by a cannon blast). Once generated by the CPU, debris primitives can be dispatched fully to the GPU for physical simulation and final rendering. Debris primitives can also interact with game-play critical objects, through an approach that provides the GPU with a one-way transfer of critical information that allows debris primitives to respond to game-play objects and large-scale world definitions.

It is important to mention that, despite the number of works devoted to GPGPU available in literature, no work deals with the issue of distribution of tasks between CPU and GPU, as proposed here.

4 GPU-CPU Process Distribution Model

For games and other real-time visualization and simulation applications, there are many different known loop models, such as the simple coupled, synchronized coupled and the multithread uncoupled [Valente et al. 2005].

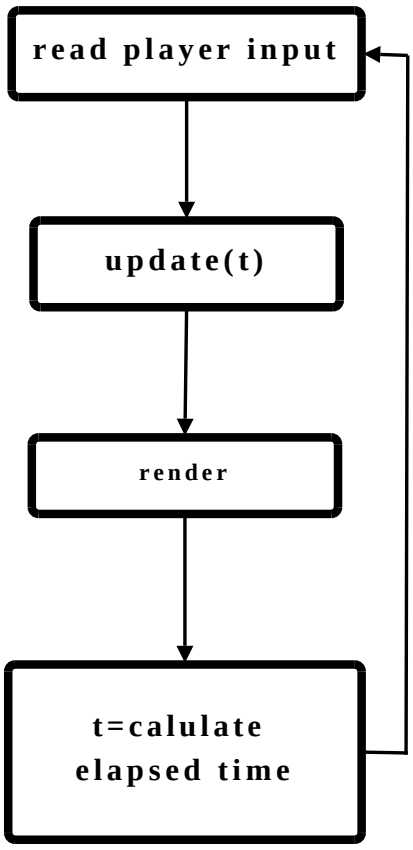

Figure 1: Simple coupled model.

Basically, these architectures arrange typical processes involved in a game in different manners, but always inside a main loop. The processes consist of the following ones: data input (from keyboard, joystick, mouse, etc.), update — especially physics, artificial intelligence (AI), and application logic — and rendering.

In the simple coupled model, the stages are arranged sequentially, as shown in Figure 1.

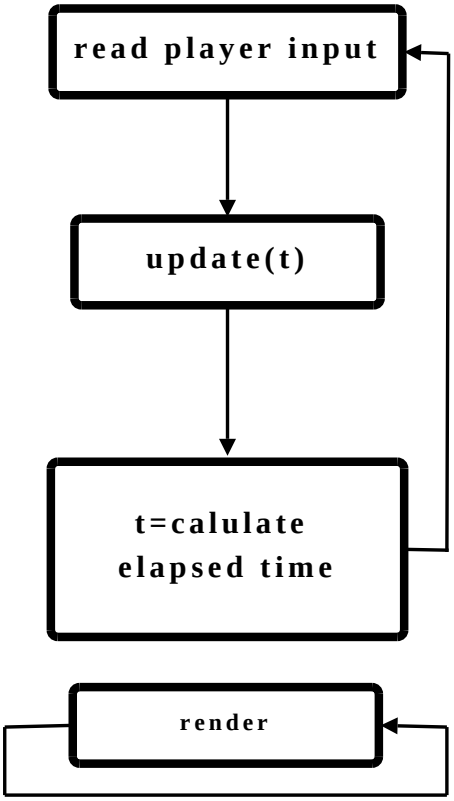

Figure 2: Multithread uncoupled model.

The multithread uncoupled model, depicted in Figure 2, separates the update and rendering stages in two loops, so they can run independently from each other. In this model, the input and update stages can run in a thread, and the rendering stage can run in another thread.

As presented before, the described models comprise three stages: input, update, and rendering. With the possibility of using the GPU for generic computation, this paper proposes a new model, called *multithread uncoupled with GPGPU*. This model is based on the multithread uncoupled model with the inclusion of a new stage, defined as GPGPU. This architecture is composed of threads, one for the input and update stages, another for the GPGPU stage, and the last one for the rendering stage. Figure 3 depicts a schematic representation for this approach [Zamith et al. 2007].

In the parallel programming models, it is necessary to detect which are shared and non-shared parts, which will be treated differently. The independent sections compose tasks that are processed in parallel, like the rendering task. The shared sections, like the GPGPU and the update stages, need to be synchronized in order 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 first stage that should be run is the update, followed by GPGPU stage, while the render stage runs in parallel to the previous ones. The update and GPGPU stages are responsible for defining the new game state. For example, the former calculates the collision response for some objects, whereas the latter defines new positions for the objects. The render stage presents the results of current game state.

The processing dependence of shared objects needs to use a synchronization object, as applications that use many threads do. Multithread programming is a complex subject, because the tasks in the application run alternately or simultaneously, but not linearly.

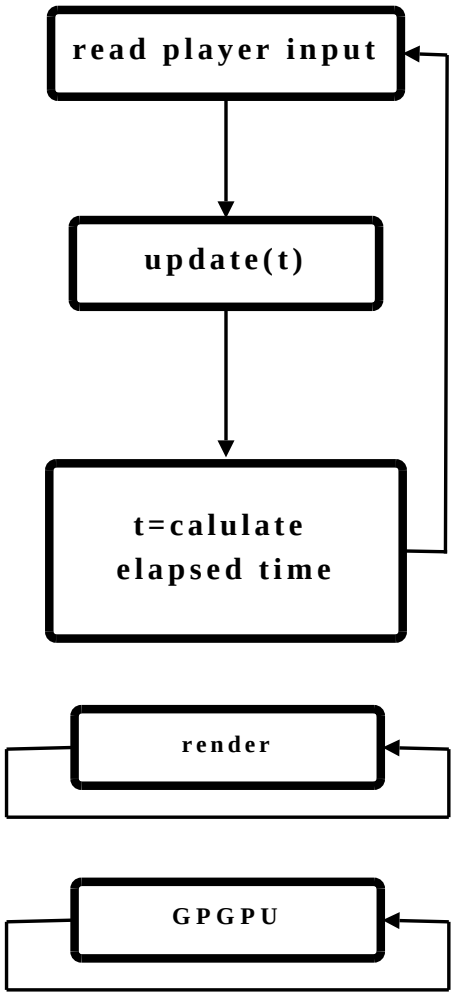

Figure 3: Multithread uncoupled with GPGPU model

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 current implementation of the presented model uses semaphores as synchronization object.

The framework architecture provides the abstract class AbstractAppRunner to run a generic application. The methods in that class represent game loop stage and other system events, that are dispatched during the application life cycle. The approach presented in this work implements the concrete class GPGPUAppRunner, derived from AbstractAppRunner, which is a composed by a main loop and two thread objects.

The thread objects are instances of two concrete classes: GameLoopThread and GPGPULoopThread. The GPGPULoopThread, which extends the abstract class AbstractThread, defines virtual methods to run the thread loop and to change the semaphores. The GPGPURunner loop runs the data input and render stages, the GameLoopThread loop runs the update stage, and the GPGPULoopThread loop runs the GPGPU stage. Figure 7 shows the UML class diagram of this architecture.

Even if the tasks should obey a predefined execution ordering, the multithread approach makes it possible to run the stages simultaneously.

The distribution model presented in this work is fixed, being the distribution of the tasks between the CPU and GPU defined previously. The implementation of a desirable automatic model of process distribution is a very intricate issue and even more if it is considered distribution between two different hardware architectures: the conventional CPU and the GPU. Besides, as it was mentioned previously, not all kinds of tasks can be processed by the GPU.

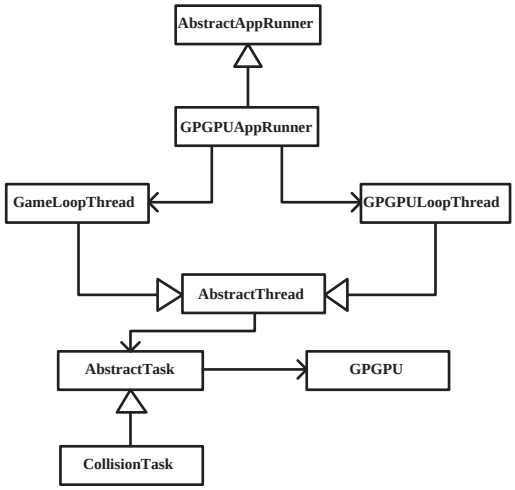

Figure 4: Main classes of the proposed model

5 Example and Results

The case study implements collision detection among moving solids. In a previous version, the collision detection process was implemented entirely on CPU, in the updated stage. The subsequent version shows how this process can be performed in the GPGPU stage.

The use of GPU for general-purpose computation requires an OpenGL extension called FBO (frame buffer object). A FBO is configured from parameters related to the storage format of textures in the GPU, the floating point precision (the floating point precision is 32bit IEEE standard), and the texel components (RGBA color channels). The concrete class GPGPU encapsulates details regarding the FBO setup and the fragment shader program. This class defines and implements methods that help complex manipulation of resources, like automatic configuration, texture transfering (uploading to GPU memory and downloading from frame buffer), and shader running.

As the GPGPU stage run tasks, it is necessary an intermediate layer where the tasks will be implemented. Therefore, an intermediate layer is built between the application (the game) and the GPGPU class. The objective of the class GPGPU is to create a communication interface between the application and the GPU, allowing tasks to be scheduled either for the CPU or for the GPU, with no interference from the developer. To represent the layer, the class AbstractTask was defined. This one holds a GPGPU object which in turn extends from the GPGPU class. The collision task, which extends AbstrackTask, was defined for implementing collision among solid objects.

Two methods of the class AbstractTask deserve special attention: execCPU and execGPU. The former is where the CPU algorithm will be implemented, and in the latter is where the GPU algorithm will be implemented. Therefore, for each task that will be processed on GPU is necessary to write a new class that extends the class AbstractTask. AbstractThread is an abstract class that implements APIs for thread creation and manipulation. The class declares a pointer to an object of a class derived from AbstractTask. GameLoopThread and GPGPULoopThread are extensions of AbstractThread; thus, GameLoopThread runs tasks on CPU by invoking execCPU task method, and GPGPULoopThread runs tasks on GPU by invoking execGPU task method (see UML class diagram in Figure 7).

The class AbstrackTask holds a vector of pointers to solid objects. The method execGPU is responsible for mapping some object properties to floating point RGBA textures. The properties are position, velocity, and acceleration. As these properties have three dimensions, the execGPU method uses three textures, one for each property. Each value in the property is mapped to a texel in the texture, so the RGB values store those values. Then, the class GPGPU uploads these textures to the GPU memory. As soon as the textures

are uploaded, the GPU will run the shader program $(n-1)$ times, where n is the number of solid objects in scene. The number of collision tests generated is

$$C_n^2 = \binom{n}{2} \tag{3}$$

After the shader program runs, the GPGPU class assembles the results in a output texture. A texel in this texture corresponds to an object that was tested. For each texel, the RGB values store the corrected object position and the alpha value indicates if a collision was found (one) or not (zero).

For example, consider a scene with sixteen objects where their positions are mapped in a texture. Figure 5 illustrates a simple representation of how shader variables (handles) are used to access texels, where each object position is mapped to a color. The first handle points to the first texel of the texture. The second handle makes it possible to access neighbouring texels by offsetting the first handle. Therefore, it is possible to access different solid objects, and then to perform collision detection among them.

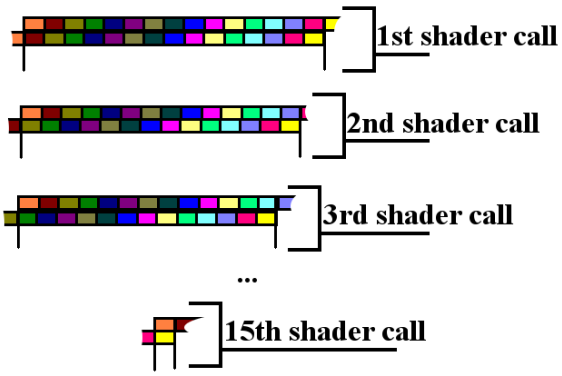

Figure 5: Representation of objects position in texture.

The parallelism between positioning camera and collision detection happens at a distinct moment. In this case, there is not parallelism among rendering, object update, and collision detection. The moving objects represent the part of application that shares resources.

In the case study, there is parallelism among player input update and object rendering, and camera positioning. Figure 6 shows which tasks are executed simultaneously and which are executed lineally. So, the render stage and camera update have even happened, but on the other hand, collision and update task have never happened together. The alternation is guaranteed by semaphor signals between the threads, that is, the signals are realized between the threads (GameLoopThread and GPGPULoopThread) that execute the updated and GPGPU stages (collision task). The synchronization is guaranteed by two variables: RedSync and GreenSync, whose values are shared, i.e., the RedSync variable of one thread shared the same value that the GreenSync variable of the other thread. Thus, the tasks are exclusively executed. After time t5 a new iteration is executed. The figure also shows how tasks are distributed between CPU and GPU: main thread runs on CPU and is responsible for camera and object update; render and GPGPU threads runs on GPU and are responsible for scene rendering and GPGPU collision detection.

Figure 7 illustrates two colliding objects. At time t0, both bodies are about to collide, and at time t1 they enter a collision state, at this moment the shader program is invoked by GPGPU thread. The shader program running on the GPU detects this collision and corrects the object positions. At time t2, their positions are updated because they are read from the FBO.

In order to evaluate the proposed model, its performance was compared with the original loop model of the framework [Valente et al. 2005]. The hardware used was a Pentium D 3.4Ghz, 1GB of memory, and a GPU NVIDIA GeForce 6200 AGP.

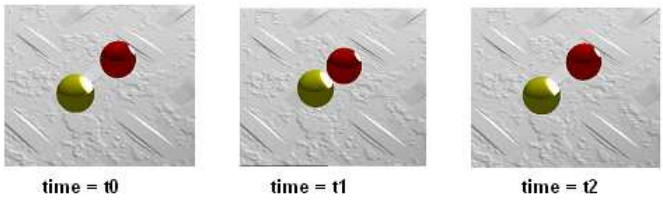

Figure 7: Collision between two spheres.

Two classes of tests were constructed using the framework. The first one uses the single coupled synchronized model and the other the multithread uncoupled with GPGPU, as proposed in this work. For each class, 10 tests were executed to measure the processing time in the GPU and the CPU. Each class consists in 500 collisions between 16 moving solids positioned in the scene in an arbitrary way, without any user interaction.

Tables 1 and 2 show the results corresponding to the multithread with GPGPU and simple coupled synchronized models, respectively. Column labeled **total col. t.** represents the time for computing the 500 collisions, **total app. t.** is the total running time of the application, **work** is the effective time in the GPU and CPU (all of them in seconds), and **FPS** the number of frames per second.

Table 1: Results: multithread uncoupled w. GPGPU model

total col. t.	total app. t.	works	FPS
1,578	0,015	0,030000014	77,3131
1,938	0,016	0,047000030	79,9794
1,703	0,016	0,062999996	76,9231
2,219	0,016	0,032000001	84,2722
1,719	0,016	0,032000004	77,9523
2,093	0,016	0,032000085	84,5676
2,234	0,016	0,016000003	83,2587
1,968	0,016	0,045999954	82,8252
1,312	0,016	0,030999969	70,8841
2,203	0,016	0,015999996	83,9764

Table 2: Results: simple synchronized coupled model

total col. t.	total app. t.	works	FPS
22,625	0,016	0,063000096	240,442
83,594	0,016	0,330000447	242,290
44,234	0,016	0,232999674	242,302
27,140	0,016	0,329000170	244,068
42,828	0,016	0,266999744	240,870
50,063	0,016	0,234000313	242,714
14,328	0,016	0,063999999	238,554
15,328	0,016	0,126000026	238,192
30,188	0,016	0,109999970	241,420
22,110	0,016	0,172999781	240,299

Table 3 represents the comparison of the mean processing times corresponding to 10 test instances. Lines labeled **FPS** represents the number of frames per second and **work** the effective processing time, respectively. Lines **time**, **minimal**, and **maximum** correspond respectively to the mean, the minimum, and the maximum processing times of the 10 instances (in seconds).

The results in Tables I, II and III show an increase in performance obtained by using the multithread uncoupled model with GPGPU. This increase in performance is due to the concurrent execution of the tasks.

Using the GPGPU component of the architecture considerably reduced the processing time of the collision detection part of the system. This caused an overall decrease in the processing time of the application. Besides, the tests have shown that the frame rate de-

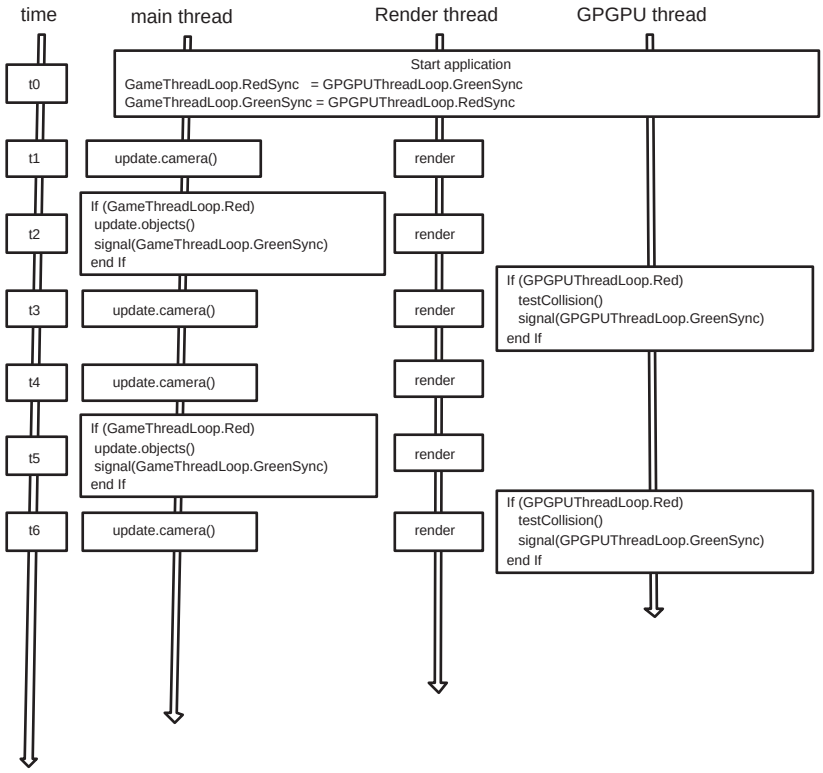

Figure 6: Task parallelism representation

Table 3: GPU and CPU comparatives

	GPU	CPU
FPS	80,1952100	241,1151000
works	0,0345000	0,1929000
time	1,8967000	35,2438000
minimal	0,0160000	0,0630001
maximum	0,0630000	0,3300004

crease, to enable GPGPU processing, did not affect the quality of the animation, which was preserved.

6 Conclusions and Future Work

Balancing the load between CPU and GPU is an interesting approach for achieving a better use of the computational resources in a game or virtual and augmented reality applications. By doing this it is possible to dedicate the additional free computational power to other tasks as AI and complex physics, which could not be done in more rigid or sequential architectures.

This work has demonstrated this possibility by introducing a new stage in a multithread uncoupled game engine architecture, which is responsible for general-purpose processing on the GPU. As it was exemplified by a solution for the collision detection problem, it is possible to use this same approach for other problems and tasks as AI and those listed in Section 2, which can be also processed in this component of the architecture by the GPU. In order to do this it suffices to do the correct mapping of the objects to the appropriate textures and design the corresponding shader responsible for implementing the algorithm associated to the solution.

The results tabled in Section 5 demonstrate that the use of GPU for general-purpose computation is a promising way to increase the performance in game engines, virtual and augmented reality systems, and other similar simulation applications.

In the most recent graphics processors, as the GeForce 8 series, it is possible to use one of its GPUs for running a thread responsible for managing the load balancing between CPU and GPU. The authors intend to pursue this direction in a future work. Another point to be investigated is how to detect in a more automatic way which processes are appropriate for CPU or GPU allocation.

References

BACIU, G., AND WONG, W. S. K. 2003. Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Computer Graphics 9*, 2, 254–271.

BOLZ, J., FARMER, I., GRISPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3, 917–924.

BOND, A. 2006. Havok FX: GPU-accelerated physics for PC games. In *Proceedings of Game Developers Conference 2006*. Available at `www.havok.com/content/view/187/77/`.

EBERLY, D. H. 2004. *Game Physics*. Morgan Kaufmann.

ERICSON, C. 2005. *Real-Time Collision Detection*. Morgan Kaufmann.

FATAHALIAN, K., SUGERMAN, J., AND HANRAHAN, P. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004*, 133–138.

FEIJÓ, B., PAGLIOSA, P. A., AND CLUA, E. W. G. 2006. Visualização, simulação e games. In *Atualizações em Informática*, K. Breitman and R. Anido, Eds. Editora PUC-Rio, 127–185. (In Portuguese).

GALOPPO, N., GOVINDARAJU, N. K., HENSON, M., AND MANOCHA, D. 2005. LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 3–14.

GOVINDARAJU, N. K., REDON, S., LIN, M. C., AND MANOCHA, D. 2003. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, 25–32.

GREEN, S., 2003. NVIDIA cloth sample. Available at `download.developer.nvidia.com/developer/`

SDK/ Individual_Samples/ samples.html# glsl_physics.

HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2004. Detection of collisions and self-collisions using image-space techniques. *Journal of WSCG 12*, 3, 145–152.

JAMES, D. L., AND PAI, D. K. 1999. Accurate real time deformable objects. In *Proceedings of ACM SIGGRAPH 99*, 65–72.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. UberFlow: a GPU-based particle engine. In *Graphics Hardware 2004*, 115–122.

KOLB, A., LATTA, L., AND RESK-SALAMA, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware 2004*, 123–132.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3, 908–916.

OWENS, J. D., LEUBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*. To appear.

RUMPF, M., AND STRZODKA, R. 2005. Graphics processor units: New prospects for parallel computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, vol. 51 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, 89–134.

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.

VALENTE, L. 2005. *Guff: um framework para desenvolvimento de jogos*. Master's thesis, Universidade Federal Fluminense. (In Portuguese).

ZAMITH, M. P. M., CLUA, E. W. G., CONCI, A., MOTENEGRO, A., PAGLIOSA, P. A., AND VALENTE, L. 2007. Parallel processing between gpu and cpu: Concepts in a game architecture. *IEEE Computer Society*, 115–120. ISBN: 0-7695-2928-3.

ZELLER, C. 2005. Cloth simulation on the GPU. In *ACM SIGGRAPH 05: ACM SIGGRAPH 2005 Sketches*.