Parallel processing between GPU and CPU: Concepts in a game architecture

Marcelo P. M. Zamith,
Esteban W. G. Clua, Aura Conci
and Anselmo Montenegro
Instituto de Computação Universidade Federal Fluminense
Rio de Janeiro, RJ, Brasil 24210-240
{mzamith,esteban,aconci,anselmo}@ic.uff.br

Paulo A. Pagliosa
Departamento de Computação e Estatística
Universidade Federal de Mato Grosso do Sul
Campo Grande, MS, Brasil 79070-900
pagliosa@dct.ufms.br

Luis Valente
Departamento de Informática
PUC-Rio
Rio de Janeiro, RJ, Brasil 22453-900
lvalente@inf.puc-rio.br

Abstract

This paper presents a new game loop architecture concept that employs graphics processors (GPUs) for general-purpose computation (GPGPU). A critical issue in this field is the concept of process distribution between CPU and GPU. The presented architecture consists in a very efficient model for distribution, and it was mainly designed to support math and physics on the GPU, but any kind of generic computation can be easily adapted.

Keywords— **game loop, GPGPU, process distribution.**

1 Introduction

GPGPU (General-purpose computation on GPUs) is a new area and recent research field related to Computer Graphics. It stems from new graphics hardware architectures, which can also be used for generic computation, thus introducing new paradigms.

However, not all kinds of generic algorithms can be processed on GPUs, due to their internal architecture. Suitable algorithms are the ones that follow the SIMD (single instruction, multiple data) approach or are implemented as a stream-based process[14]. This is the case of many math and physics simulation algorithms. Another issue relates to memory latency access to GPU memory, that directly affects GPU performance. Then, even though some algorithms are suitable for running on the GPU, if it requires frequent memory accesses its performance may be severely degraded.

A good process distribution policy should strive to maintain an identical load of math, physics, or other GPGPU problems for the CPU and the GPU to solve. Current graphics hardware are able to generate more than 60 frames per second. In spite of that, users cannot perceive differences in image or animation quality for frame rates above this value. Thus, it is a good idea to limit the frame rate and use the remaining computation power on the GPU to solve other kinds of problems. Also, as it is not always possible to tell which processing route (GPU or CPU) a problem should go through, the framework or engine being used for application development should be responsible for allocating jobs dynamically.

This paper proposes a new architecture concept for a correct and efficient process load distribution, considering real time applications like games or virtual reality environments.

The paper is organized as follows. Section 2 summarizes the main functionalities of a physics engine for games. 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 concepts related to the proposed model and how processing distribution between CPU and GPU is done. Finally, Section 5 points out conclusions and future works.

2 Math and Physics for Games

The ODE (Open Dynamics Engine) is an open source component that is responsible 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.

Generally, the motion of a rigid body is not free, but

Computer Graphics, Imaging and Visualisation (CGIV 2007)
0-7695-2928-3/07 $25.00 © 2007 IEEE

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 [6].

In order to compute the contact forces that will prevent inter-penetration 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 [5]. 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 [4].

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 [5]. 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 [1, 7, 9].

The ODE solver is a component that can be also efficiently implemented on GPU, since the integration numeric for a rigid body is performed independently of the other ones (therefore in parallel). 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.

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 [15], 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 [10]. 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.

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 [13]. This section cites works related to math and physics of solids on GPU.

Bolz et al. [2] 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 [11]. 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.

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 [8, 18].

Recently, NVIDIA and Havok have been shown that rigid body simulations for computer games perform very well on GPUs [3]. 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 pre-modeled 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 [17].

Basically, these architectures arrange typical processes involved in a game as a main application loop. For example, process can be: 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 1 also depicts the multithread uncoupled model, that 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.

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 2 depicts a conceptual representation for this approach.

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. 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 object, whereas the latter defines new positions for the objects. The render stage presents the results of the current game state. Because the stages depend on each other, it is necessary to synchronize them explicitly using synchronization objects, like semaphores.

4.1 The Architecture

In order to realize an architecture for the GPU-CPU Process Distribution Model, it is necessary to implement a game loop and to handle system events that are dispatched during the application life cycle. This work presents an approach that implements a game loop as a main loop with two ancillary threads. The main loop runs the update stage, one thread runs the rendering stage whereas the other thread runs the GPGPU stage.

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

The process distribution model in this work is static, meaning that the tasks are scheduled previously for the CPU and GPU through a Lua script file[12]. An implementation of an automatic process distribution model is very intricate due to the very different nature of the involved hardware architectures. Besides that, not all kinds of tasks can be processed by the GPU, as mentioned previously. This model is designed for tasks that are able to run on both the CPU and the GPU. However, to understand it fully, it is necessary to introduce two concepts, tasks and task management.

4.2 Threads

The multicore technology featured in new processors and game consoles makes it more efficient to run multithread applications, because the threads can run in different cores simultaneously. Hence, developing multi-thread applications is becoming more relevant, and it is important to understand certain aspects of this development approach.

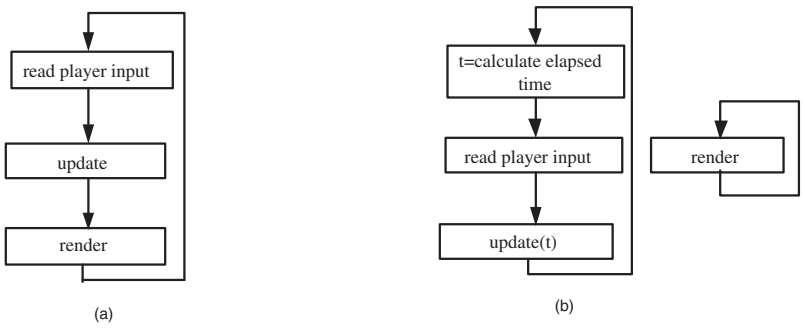

Figure 1: Game loop models. (a) Simple coupled model (b) Multithread uncoupled model

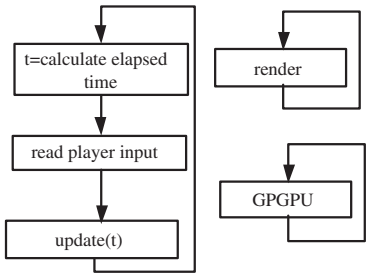

Figure 2: Multithread uncoupled with GPGPU model

Whenever different threads run simultaneously, their intructions are executed in parallel and not linearly, as in conventional applications. Problems arise, for example, when the threads need to share information of have some processing dependence. In these cases, it is necessary to synchronize the threads.

Access to shared information must be serialized so the threads do not interfere with each other. Consequently, when a thread is working on a shared area, the other threads must wait for their turn to enter it. There are a number of solutions to use for implementing mutual exclusion among threads. The architecture this work describes adopts the semaphore as the mutual exclusion object. Other solutions could be easily applied.

Another important property of threads is the priority. The priority determines how often a thread will be scheduled to run, that is, a thread with higher priority will run more often than another that has lower priority [16]. Real-time tasks demand threads with high priority, while other tasks can run with lower priorities.

There are five states a thread can be in: allocating, destroying, suspending, resuming, and running. A new thread is in the allocating state. The operating system is responsible for fulfilling this task, when it allocates the thread context. The thread context corresponds to memory space, kernel-thread object, and program counter, among other structures. A thread is in the suspending state when it is execution is paused. Resuming is the state a thread is in when it is requested to run again, after being suspended. Destroying is the state a thread is in when it should stop completely, and its resources return to the operating system. It is important to observe two subtleties when working with threads. The first one is that suspending a thread is not enough to synchronize it with others [16], that is, it is not possible to know exactly at which intruction a thread stopped after the suspension request. As a consequence, inconsistency problems may occur when the thread is resumed [16]. The second subtlety is that a thread must be synchronized with others before it is destructed, or errors may arise when the applications shuts down.

4.3 Tasks and Processors

The presented architecture considers tasks as units of execution, representing a specific functionality. The architecture makes it possible to tasks run on the CPU or the GPU. Hence, a consequence of this design is that tasks could run simultaneously on the CPU and the GPU. Hence, it is possible that the task runs on the CPU or the GPU simultaneously. This behavior is not desired and to hinder it, the tasks have a flag that indicates on which processor it is running.

The other constraint the architecture enforces is to guarantee mutual exclusion among threads. This situation arises when two interdependent tasks run on different processors. When different tasks always run on the same processor, this constraint is not enforced because they run linearly.

4.4 Management of Shared Tasks

The architecture has a task manager that is responsible for allocating tasks to processors. The initial scheduling is specified by the user through a Lua script file. The task manager then interprets the script and setups the tasks. There is only one instance of the task manager through the application life cycle. The reason for this approach is to avoid synchronization and consistency problems in the multi-thread architecture.

The main purpose of the task manager is to supply tasks to threads. Whenever a thread (CPU or GPU thread) starts running, it requests a task to the task manager. When this task is over, the task manager will provide the next one to run. If there are not tasks to run, the requesting thread will be suspended.

When writing the configuration script, the user must define which tasks are going to run on each processors, and their ordering. For example, a problem may have three tasks A, B, and C, and the user may inform that tasks A and B should run on the GPU and task C should run on the CPU. Also, if it is required that task B should run before task A, then the user will also specify this contraint in the script file. It is important to notice starvations and deadlocks may manifest themselves under multi-thread applications, hence the user should be careful when planning the scheduling, so they can be avoided.

5 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.

Particles simulation and rigid body collision detection are tasks that can be implemented easily on the GPU. Collision detection among rigid bodies was the task chosen to validate the proposed model. Processor time after 500 collision was 1.8967000 seconds for the GPU versus 35.2438000 seconds for the CPU. Hence, the GPU surpasses the CPU when processing time is considered.

This work has shown the possibility by introducing a new stage in a multithread uncoupled game engine architecture, which is responsible for general-purpose processing on the GPU.

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

[1] G. Baciu and W. S. K. Wong. Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):254–271, 2003.

[2] J. Bolz, I. Farmer, E. Grispun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.

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

[4] D. H. Eberly. *Game Physics*. Morgan Kaufmann, 2004.

[5] C Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[6] B. Feijó, P. A. Pagliosa, and E. W. G. Clua. Visualização, simulação e games. In K. Breitman and R. Anido, editors, *Atualizações em Informática*, pages 127–185. Editora PUC-Rio, 2006. (In Portuguese).

[7] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, pages 25–32, 2003.

[8] S. Green. NVIDIA cloth sample. Available at `download.developer.nvidia.com/developer/SDK/ Individual_Samples/samples.html#glsl_physics`., 2003.

[9] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. *Journal of WSCG*, 12(3):145–152, 2004.

0-7695-2928-3/07 $25.00 © 2007 IEEE

[10] D. L. James and D. K. Pai. Accurate real time deformable objects. In *Proceedings of ACM SIGGRAPH 99*, pages 65–72, 1999.

[11] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[12] Lua. The programming language lua. Available at `http://www.lua.org/`, 2007.

[13] J. D. Owens, D. Leubke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 2007. To appear.

[14] Fernando R. *Programming Techniques, Tips, and Tricks for Real-Time Graphics (Hardcover)*. Addison-Wesley, 2005.

[15] M. Rumpf and R. Strzodka. Graphics processor units: New prospects for parallel computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 89–134. Springer-Verlag, 2005.

[16] A. Silberchatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons. Inc, 2005.

[17] L. Valente, Conci A., and B. Feijó. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, pages 89–99, 2005.

[18] C. Zeller. Cloth simulation on the GPU. In *ACM SIGGRAPH 05: ACM SIGGRAPH 2005 Sketches*, 2005.