

Overview

Problem

The discovery of extrasolar planetary systems has led to many surprises and reinvigorated the field of planetary orbital dynamics. Direct numerical simulations of planetary systems are necessary to interpret extrasolar planet detections and to study the formation and evolution of planetary systems in general. Due to measurement uncertainties and the highly chaotic orbital evolution of systems with three or more massive bodies, large ensembles ($\sim 10^3 - 10^7$) of planetary systems must be simulated to investigate the outcomes in a statistical sense.

Challenge

Previous research has demonstrated that GPUs can greatly accelerate a numerical simulation of the **gravitational N-body problem** for large N (e.g., star clusters). For a typical simulation of planetary system, $N \simeq 3 - 10$ (i.e., a star and 2-9 planets). Therefore, simply using the GPU to calculate the particle accelerations (as is practical for large N -body simulations) is extremely inefficient due to the latency in memory transfers between the CPU and GPU. In order to be efficient on a GPU, we must implement the majority of the integrator directly on the GPU.

Conclusions

We have begun to investigate whether GPUs can significantly accelerate the integration of a large ensemble of few-body simulations. Here we report that a second-order, time-symmetric, adaptive time-step integration algorithm can integrate ensembles of $\geq 4,028$ planetary systems roughly 100 times more efficiently than the a uniprocessor version of the same integration algorithm using a comparable speed CPU. We are now attempting to implement a higher order integrator and to optimize our algorithms further.

Our Algorithm

Initial Conditions

Each planetary system consists of a star with the mass of the Sun and one or two planets with the mass of Jupiter. Each body has a mass (m_i), position (x_i), and velocity (v_i). Here we choose initial conditions for the planets such that the systems will not undergo close encounters. Thus, we can assume that there are no collisions and each body's mass is constant throughout each simulation.

Equations of Motion

All bodies interact via gravity, so their trajectories are described by **Newton's Laws of Motion** and **Newton's Law of Gradation**:

$$v_i = \frac{dx_i}{dt} \quad \text{and} \quad a_i = \frac{dv_i}{dt} = \sum_{j \neq i} \frac{m_j \times (x_j - x_i)}{|x_j - x_i|^3}.$$

Simulation Algorithm

In an effort to minimize the memory requirements and complexity of the integrator, we apply the **Verlet** method (i.e., Time-Symmetric Adaptive Time-step Leapfrog) to integrate the above equations of motion. The j th time step consists of:

- First half of a **drift step**: $x_{j+1/2} = x_j + v_j \Delta t_{j-1}/2$
- Updating the time step: $\Delta t_j = [(g(x_{j+1/2})\Delta\tau)^{-1} - (\Delta t_{j-1})^{-1}]^{-1}$
- A **kick step**: $v_{j+1} = v_j + a_j(x_{j+1/2})(\Delta t_{j-1} + \Delta t_j)/2$
- Second half of a **drift step**: $x_{j+1} = x_{j+1/2} + v_{j+1} \Delta t_j/2$

The constant τ sets the normalization for the size of time steps. Based on physical intuition for the 2-body problem, we adopt

$$g(x) = \left[\left(1 + \left(\sum_{j \neq i} \frac{m_i m_j}{|x_j - x_i|^3} \right) \right) \right]^{-1}.$$

GPU/CUDA Configuration

To integrate a large ensemble of planetary systems, we use CUDA to group the computations as follows:

- Each **thread block** is responsible for the simulation of a **configurable number of independent planetary systems**.
- Each **thread** is responsible for **simulating one planetary system**.
- **Global memory** is used for transferring initial data from CPU only once.
- All simulation computations are done using the **shared memory** of each multiprocessor (even though no data is shared between threads). We find this provided a large performance advantage, and gives about 8-fold speedup.

System Configuration

- CPU: AMD Athlon 64 X2 Dual Core Processor 3800+ 997MHz
- GPU: nVidia GeForce 8800 GTX

Results

Performance

Table 1 Compute Times for Two-Body Planetary Systems

# of Systems	Time				
	16	32	64	128	256 (systems/block)
1024	899ms (0.878ms)*	960ms (0.938)	972ms (0.949ms)	988ms (0.965ms)	
2048	1648ms (0.805ms)	1500ms (0.732ms)	1429ms (0.698ms)	1430ms (0.698ms)	**
4096	3152ms (0.770ms)	2782ms (0.679ms)	2694ms (0.658ms)	2680ms (0.654ms)	
8192	6179ms (0.754ms)	5365ms (0.655ms)	5354ms (0.654ms)	5300ms (0.647ms)	

Table 2 Compute Times for Three-Body Planetary Systems

# of Systems	Time				
	16	32	64	128	256 (systems/block)
1024	2897ms (2.829ms)*	1980ms (1.934)	1990ms (1.944ms)		
2048	4310ms (2.104ms)	4220ms (2.060ms)	3690ms (1.802ms)	**	**
4096	8832ms (2.156ms)	7095ms (1.732ms)	7257ms (1.772ms)		
8192	15177ms (1.853ms)	14407ms (1.759ms)	14209ms (1.734ms)		

* For each pair of entries, the top time is the total compute time, and the lower time (in parentheses) is the compute time per system.

** Not possible due to limitations of GPU's shared memory

We report the wall clock times for integrating each system for **100,000** actual simulation time steps. For each row (fixed number of planetary systems), we **highlight** the choice of number of systems per thread block that achieved the maximum performance.

The position and velocity of each body was recorded every 100 steps for possible comparison or output. To show the actual efficiency of CUDA, this output data was transferred back to CPU, but not written to disk. For reference, an unoptimized version of the same algorithm being executed on a single core of a standard CPU requires $\simeq 70$ ms/system for a two-body system, and $\simeq 150$ ms/system for a three-body system.

Interpretation

- The data in tables 1 & 2 show that our CUDA-based algorithm executed about **100 times more efficiently** on a GeForce 8800GTX than the same algorithm using a single core Athlon 64 XP2 3800+ CPU.
- A large number ($\geq 2,048$) of planetary systems (and hence threads) was necessary to achieve high performance. The maximum observed performance occurred when integrating 4,096 planetary systems in parallel.
- Once there are enough thread blocks to utilize all the multiprocessors efficiently, it is better to increase the number of threads while holding the number of thread blocks constant.
- Once the GPU is nearly fully occupied, we observe little gain by further increasing the number of threads per block.

Alternate Algorithm

As part of a preliminary investigation, we investigated an alternative algorithm.

- We used the same Verlet integration algorithm. Each thread corresponded to a single body (rather than a single planetary system). This increased the granularity of computations.
- This resulted lower performance: only $\simeq 5$ times faster than our uniprocessor algorithm.
- We speculate that overhead associated with each thread caused the suboptimal performance; the key element to get best performance is to balance the granularity.

Ongoing Work

We initiated this project to test the efficiency of GPUs for performing small-n-body integrations. In particular, we hope this will be useful for studying the orbital dynamics of planetary systems. Our preliminary results show that a large speed-up is possible and have encouraged us to extend this project. We are currently working to implement and test:

- Higher-order integration algorithms to allow for larger step-sizes. We aim to determine whether algorithm complexity or memory will limit such integrators.
- A self-organized buffer using GPU's device memory to minimize the time necessary for transferring data (planet positions and velocities) back to the CPU.
- Performing initial data reduction on the GPU, so that less data needs to be transferred back to the CPU.
- Optimizations using the constant/texture cache to provide more low-latency memory.