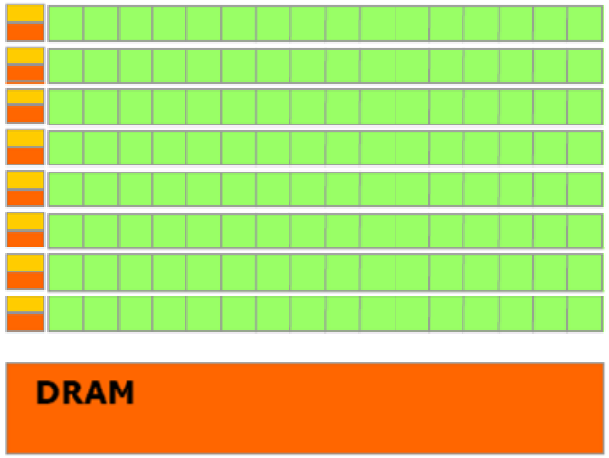
GPU-based Tools for Computational Astrophysics: N-Body Integrators for Planetary Systems

Mario Juric[1],
Eric Ford[2] (PI), Jorg Peters[2],

Aaron Boley[2], Young In Yeo[2],
(Ameya Ganchha[2], Jianwei Gao[2])

(1) Hubble Fellow, Harvard University
(2) University of Florida

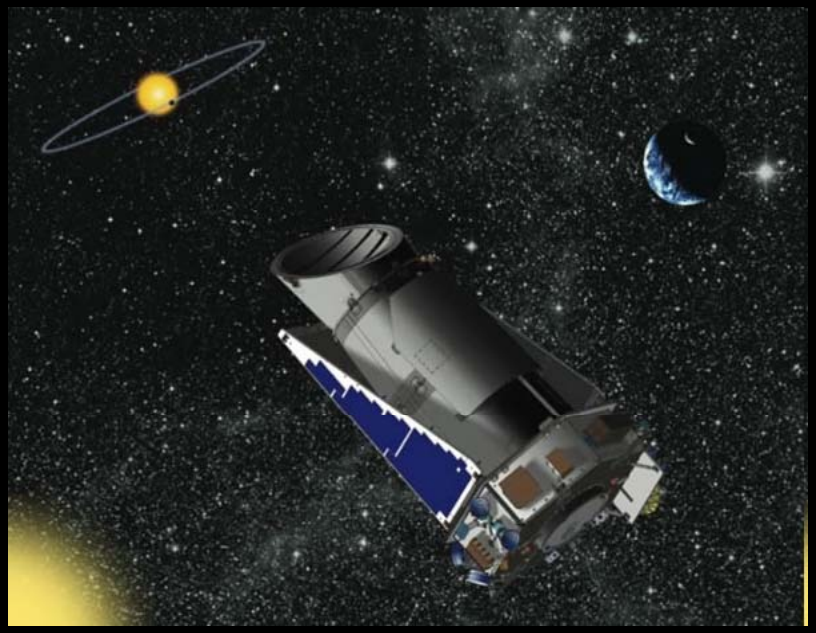


Overview

- Motivation:
  - Computational challenges of extrasolar planet characterization
  - Long-term few-body dynamics
- GPU Computing: Many-core computing of tomorrow, today
- Progress so far: Initial tests and benchmarks
- Moving forward: towards a GPU algorithm toolbox

Motivation

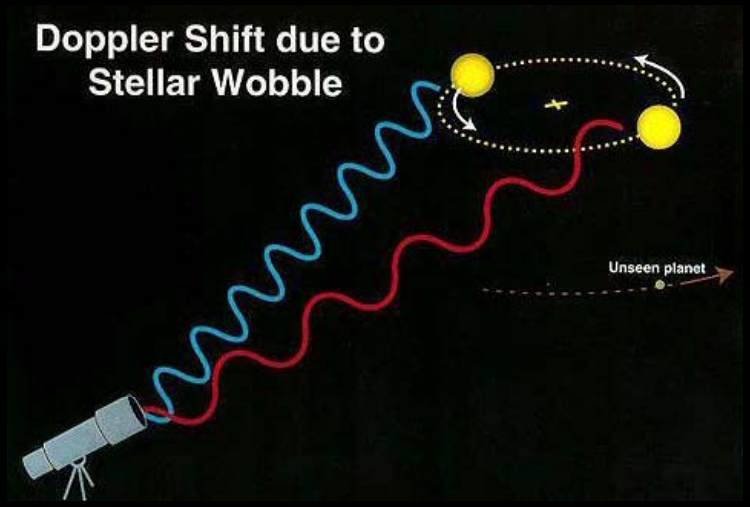
- The discovery of extrasolar planetary systems is providing insights into the formation of our solar system and humanity's place in the universe.

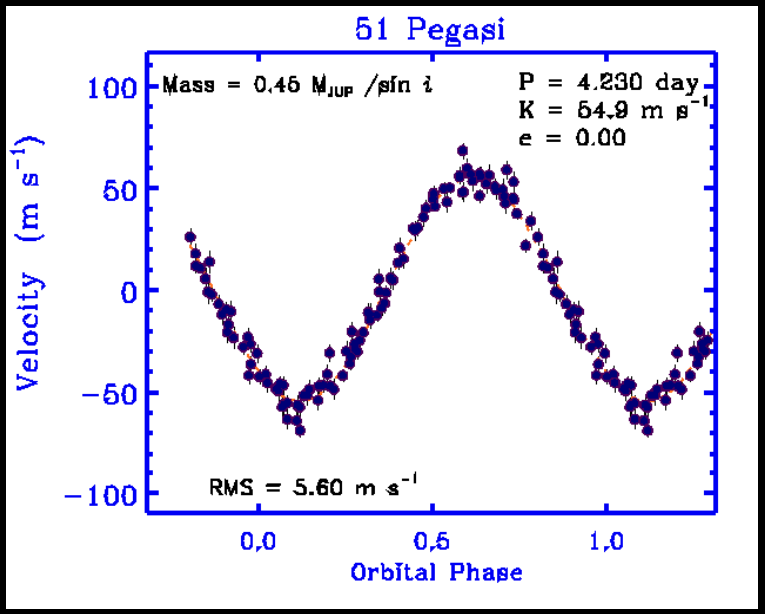


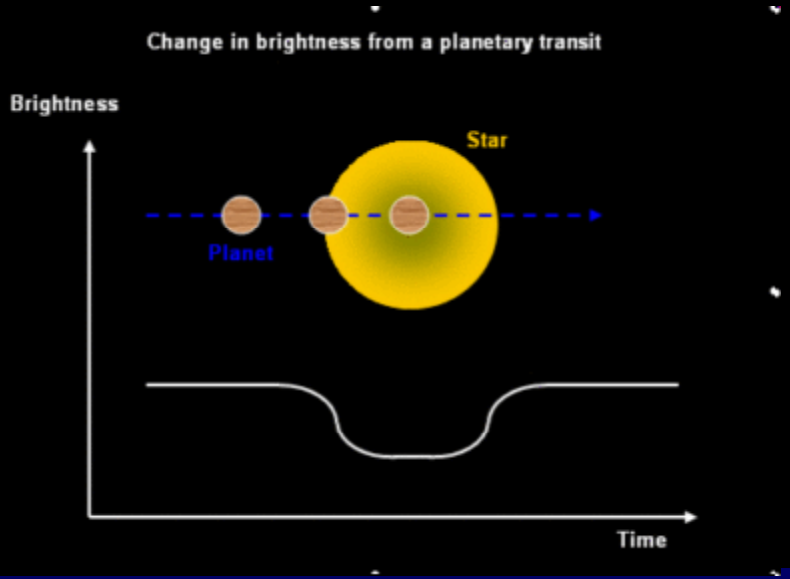
Kepler Mission: A Search for Habitable Planets (NASA)

- Detecting and characterizing multiple planet systems is particularly challenging, because accurately modeling such a system with several highly non-linear parameters is extremely computationally demanding.

Discovering extrasolar planets



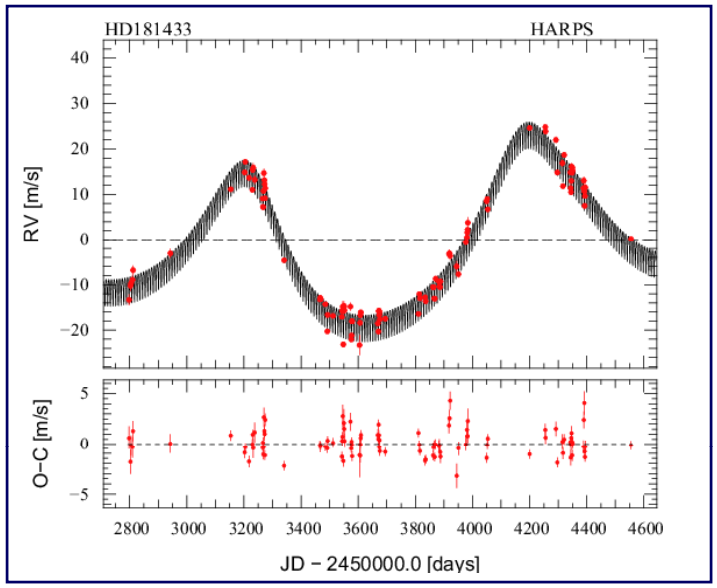


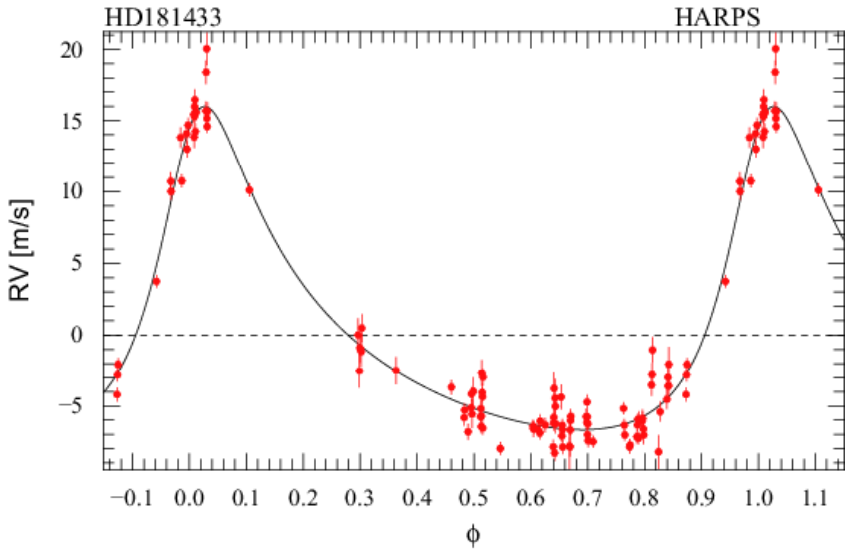


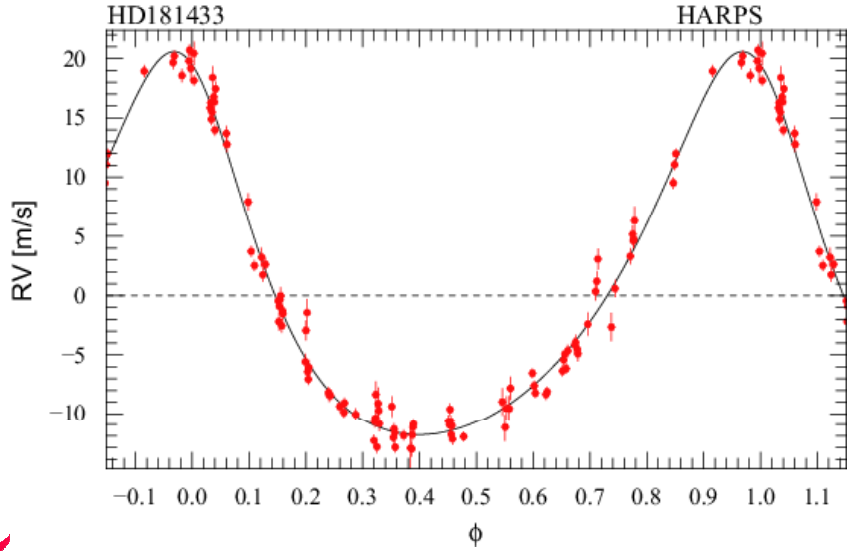
Planet(s) are nearly never observed directly; they're inferred from their (cumulative) effect on the host star.

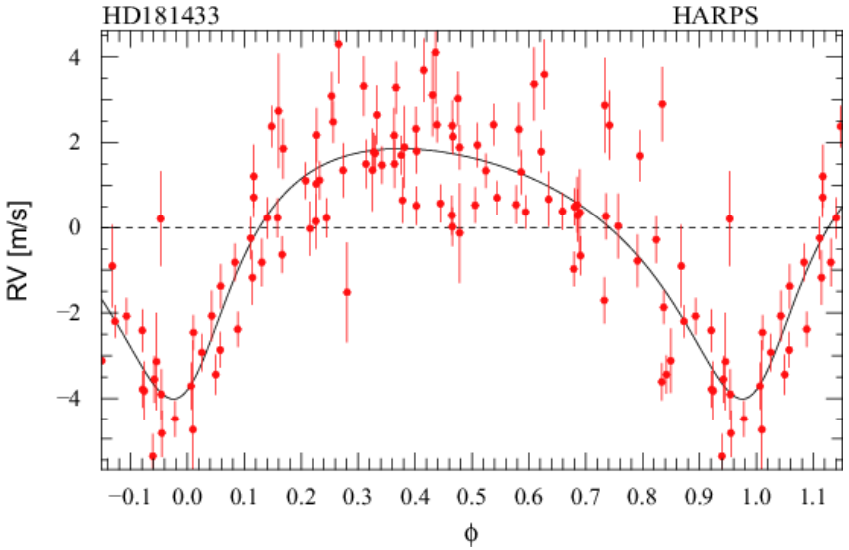
A Three-planet System

A 21-parameter fit! (Bouchy et al. 2009)









The problem: Efficient few-body/many systems integration

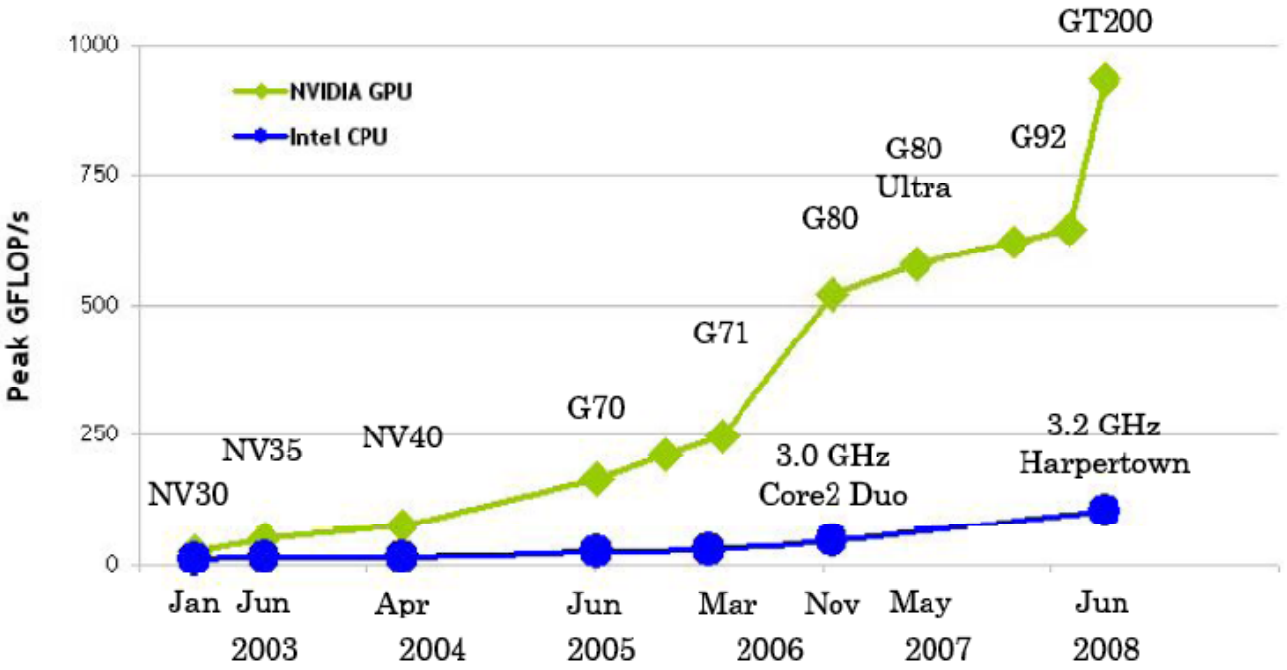
- Fitting potentially interacting planetary systems requires a full N-body integration to compute the RV curve of a trial system
- Obtaining well sampled posteriors requires ~10^6 trials (N-body integrations)
- Optimally, this procedure would be automated, with estimates updated whenever a new observation is obtained

- The bottleneck: mathematically intensive N-body integrations of millions of few-planet system

Throw more CPUs at the problem, or…

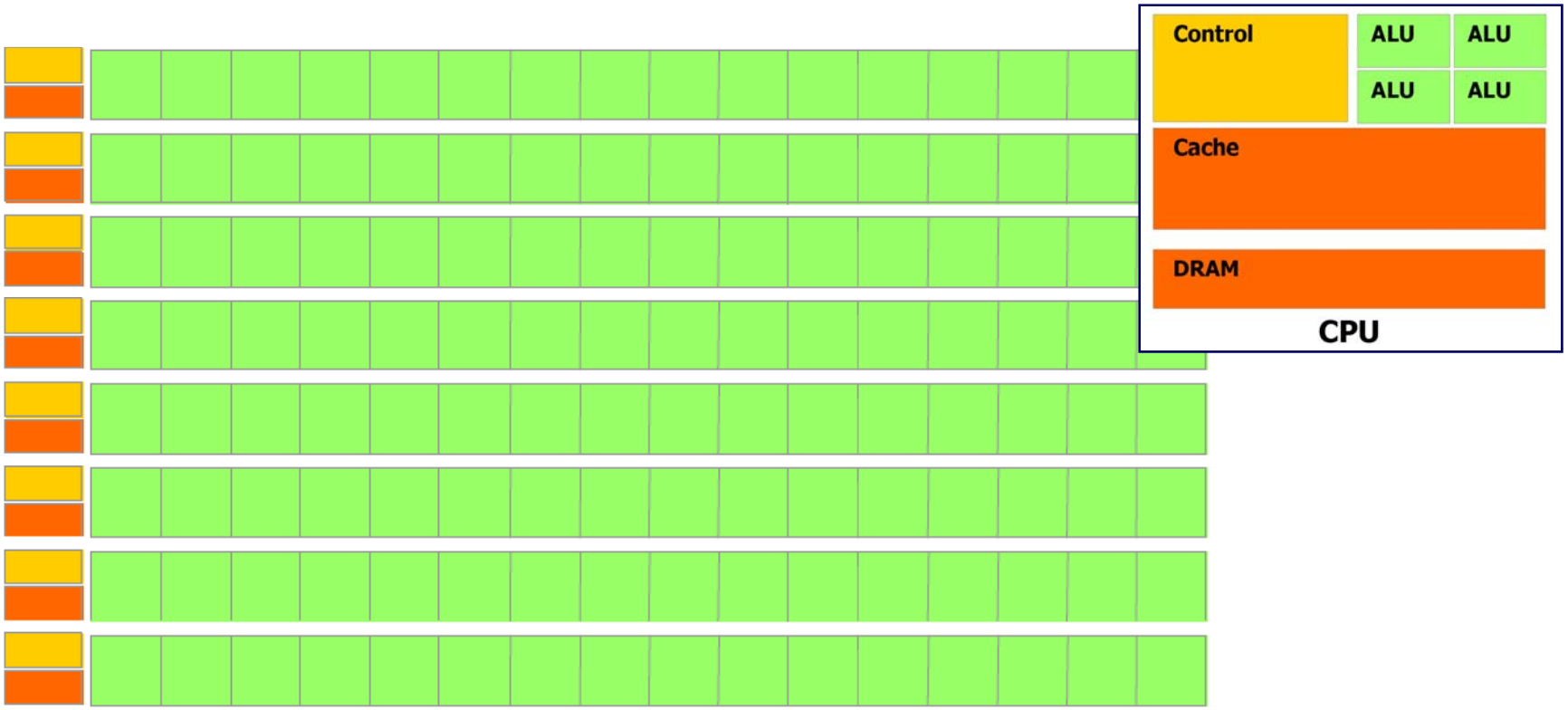
NVIDIA GeForce GTX 280 Graphics Card (~$400)

GPUs as Scientific Computing Engines



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

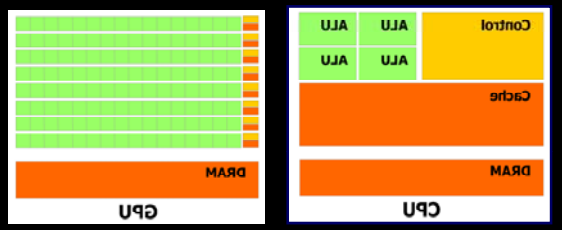
GPU: Massively Multi-core Processor



- *Hundreds of simple cores (ALUs)*
- *Minimal program control logic*
- *Exposed memory hierarchy*
- *Zero-overhead thread context switching*

GPU Design: Graphics Processing Units Heritage

- All cores execute the same program on different data
  - Process large streams of data with small, independent, arithmetic intensive, programs
    - E.g., transforming the pixels of an image
    - Transforming geometry in 3D games
    - Shading polygons 3D games



- Lots of raw computational power
  - 80% of GPU transistors are devoted to math
  - Fast basic arithmetic (single precision; DP ~8x slower)
  - Hardware implementation of common transcendental functions (sin, cos, exp, ln, …)

- Excellent overlap with needs of typical scientific codes

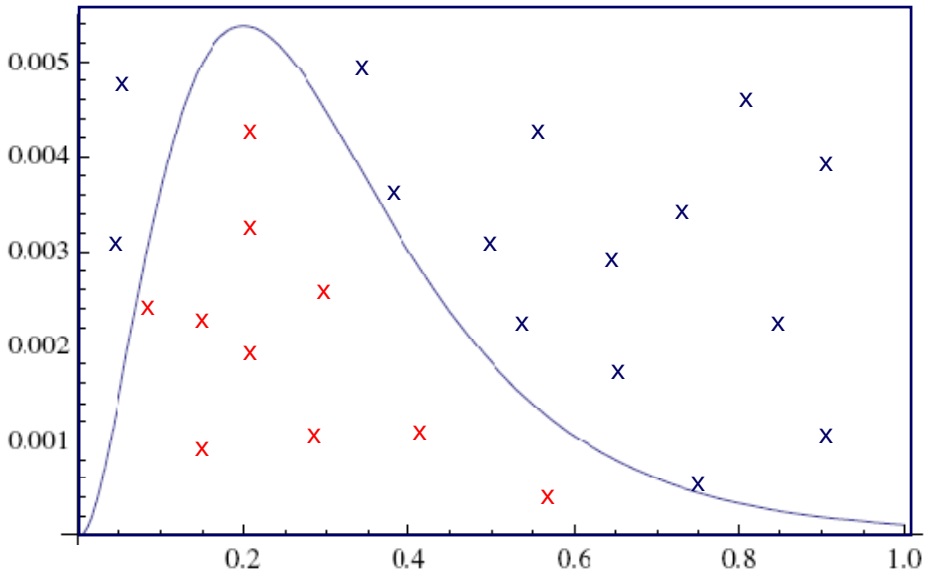
Challenges

- Not transparent for the programmer!
  - Explicitly write code that can execute in hundreds to thousands of simultaneously running threads
  - Explicitly take care of GPU memory hierarchy: manually load/store data into the cache
  - Think about “lower level” issues: caches, latencies, bank conflicts, etc...
- => GPUs are somewhat more difficult to program than traditional (single-core) CPUs.
  - But not by much!

Toy example: Monte Carlo Integration

```
In[2]:= Plot[Cos[x] * x * x * Exp[-10 * Sin[x]], {x, 0, 1}]
```

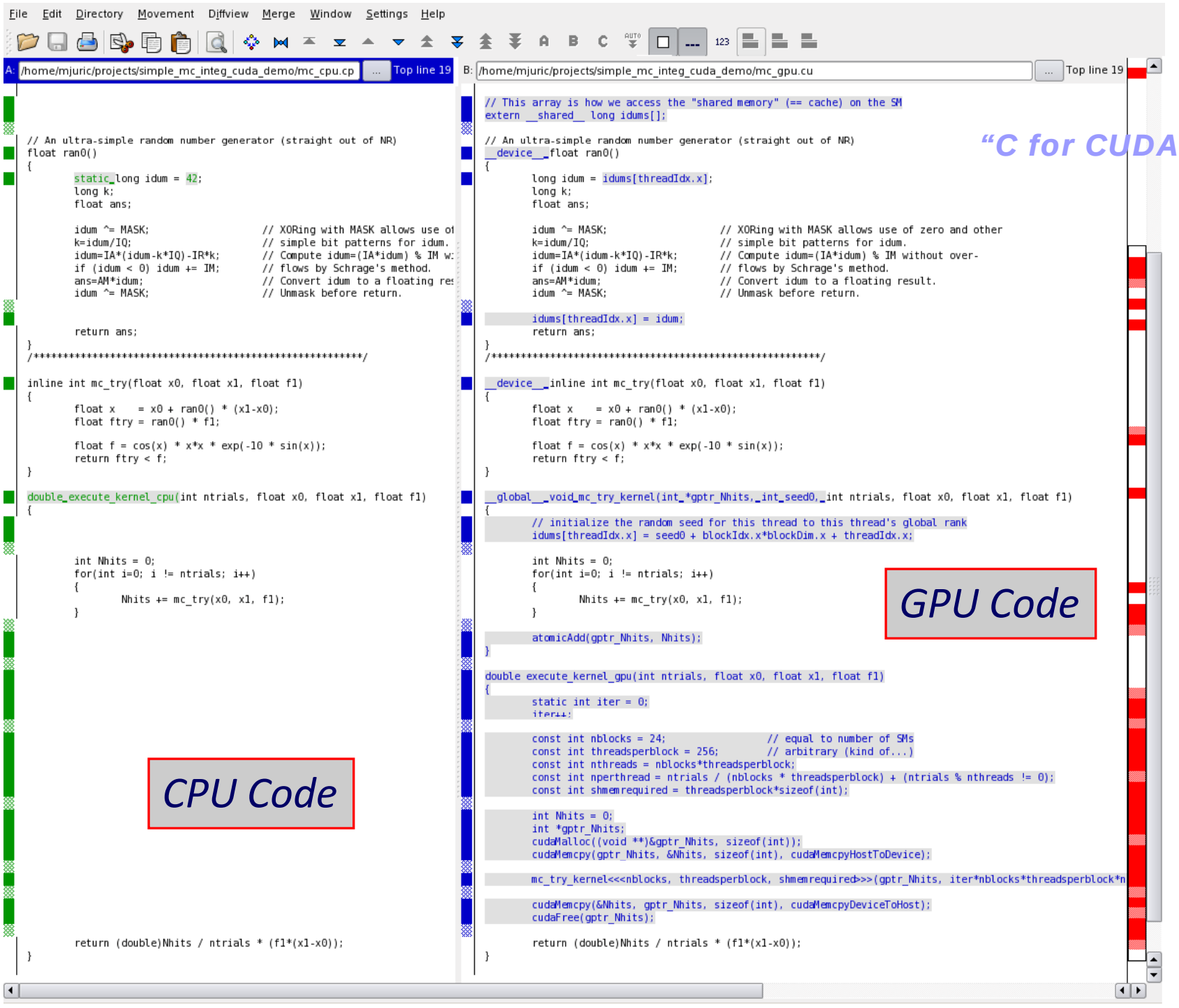
Out[2]=




```
In[4]:= NIntegrate[Cos[x] * x * x * Exp[-10 * Sin[x]], {x, 0, 1}]

Out[4]= 0.00206656
```

Area under the curve ≈
Area of the bounding rectangle × Points that fell below the curve / Number of points thrown


File Edit Directory Movement Diffview Merge Window Settings Help
A: /home/mjuric/projects/simple_mc_integ_cuda_demo/mc_cpu.cp
Top line 19
B: /home/mjuric/projects/simple_mc_integ_cuda_demo/mc_gpu.cu
Top line 19
// An ultra-simple random number generator (straight out of NR)
float ran0()
{
static long idum = 42;
long k;
float ans;
idum ^= MASK; // XORing with MASK allows use of zero and other
k=idum/IQ; // simple bit patterns for idum.
idum=IA*(idum-k*IQ)-IR*k; // Compute idum=(IA*idum) % IM without over-
if (idum < 0) idum += IM; // flows by Schrage's method.
ans=AM*idum; // Convert idum to a floating result.
idum ^= MASK; // Unmask before return.
return ans;
}
/**/
inline int mc_try(float x0, float x1, float f1)
{
float x = x0 + ran0() * (x1-x0);
float ftry = ran0() * f1;
float f = cos(x) * x*x * exp(-10 * sin(x));
return ftry < f;
}
double execute_kernel_cpu(int ntrials, float x0, float x1, float f1)
{
int Nhits = 0;
for(int i=0; i != ntrials; i++)
{
Nhits += mc_try(x0, x1, f1);
}
return (double)Nhits / ntrials * (f1*(x1-x0));
}
CPU Code
// This array is how we access the "shared memory" (== cache) on the SM
extern __shared__ long idums[];
// An ultra-simple random number generator (straight out of NR)
__device__ float ran0()
{
long idum = idums[threadIdx.x];
long k;
float ans;
idum ^= MASK; // XORing with MASK allows use of zero and other
k=idum/IQ; // simple bit patterns for idum.
idum=IA*(idum-k*IQ)-IR*k; // Compute idum=(IA*idum) % IM without over-
if (idum < 0) idum += IM; // flows by Schrage's method.
ans=AM*idum; // Convert idum to a floating result.
idum ^= MASK; // Unmask before return.
idums[threadIdx.x] = idum;
return ans;
}
/**/
__device__ inline int mc_try(float x0, float x1, float f1)
{
float x = x0 + ran0() * (x1-x0);
float ftry = ran0() * f1;
float f = cos(x) * x*x * exp(-10 * sin(x));
return ftry < f;
}
__global__ void mc_try_kernel(int *gptr_Nhits, int seed0, int ntrials, float x0, float x1, float f1)
{
// initialize the random seed for this thread to this thread's global rank
idums[threadIdx.x] = seed0 + blockIdx.x*blockDim.x + threadIdx.x;
int Nhits = 0;
for(int i=0; i != ntrials; i++)
{
Nhits += mc_try(x0, x1, f1);
}
atomicAdd(gptr_Nhits, Nhits);
}
double execute_kernel_gpu(int ntrials, float x0, float x1, float f1)
{
static int iter = 0;
iter++;
const int nblocks = 24; // equal to number of SMs
const int threadsperblock = 256; // arbitrary (kind of...)
const int nthreads = nblocks*threadsperblock;
const int nperthread = ntrials / (nblocks * threadsperblock) + (ntrials % nthreads != 0);
const int shmemrequired = threadsperblock*sizeof(int);
int Nhits = 0;
int *gptr_Nhits;
cudaMalloc((void **)&gptr_Nhits, sizeof(int));
cudaMemcpy(gptr_Nhits, &Nhits, sizeof(int), cudaMemcpyHostToDevice);
mc_try_kernel<<<nblocks, threadsperblock, shmemrequired>>>(gptr_Nhits, iter*nblocks*threadsperblock*n
cudaMemcpy(&Nhits, gptr_Nhits, sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(gptr_Nhits);
return (double)Nhits / ntrials * (f1*(x1-x0));
}
GPU Code
“C for CUDA”

AISRP Project (PI Ford)

- Goal: Write, benchmark, document, and release a set of GPU primitives for solving ODEs, with application to N-body integrators and planetary systems
- Started July 1st this year
- Plan:
  - Year 1: Assemble team, acquire GPU programming know-how, implement simple Verlet, Hermite and symplectic schemes, release initial versions
  - Year 2: Generalize to arbitrary number of bodies/complex ODEs, research feasibility of multi-precision math, optimize, document and release demonstration codes
  - Year 3: RK and BS kernels, time-symmetrized integrators, implicit methods, code cleanup and final release.

Status and First Results

- Assembling the team
  - Eric Ford (PI, University of Florida)
  - Jorg Peters (Co-I, UF CS Dept.) advising CS students
  - Young In Yeo (GS) leading the coding efforts
  - Dr. Aaron Boley (PD) just joined us (as of this week!)
  - Co-I Juric (Harvard) in extended visit to UF Jan 2010 (symplectic codes)

- Highly encouraging initial results
  1. Fast random number generator library
  2. Experimental Hermite integrator for the GPU

Hermite integration scheme for $1/r^2$ systems

Makino (1991)

- Basic step

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})dt + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(dt)^2$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})dt + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(dt)^2$$

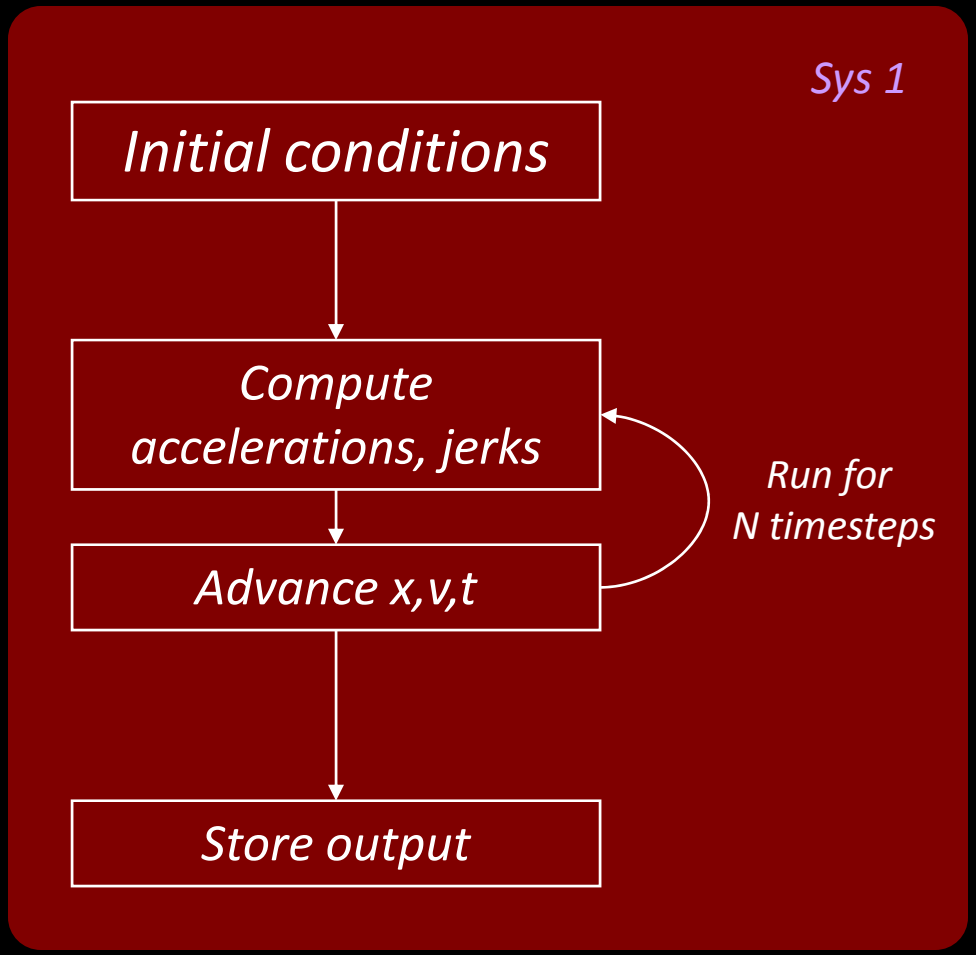
where $\mathbf{j} = \frac{d^3}{dt^3}\mathbf{r}$ is the "jerk"

or explicitly

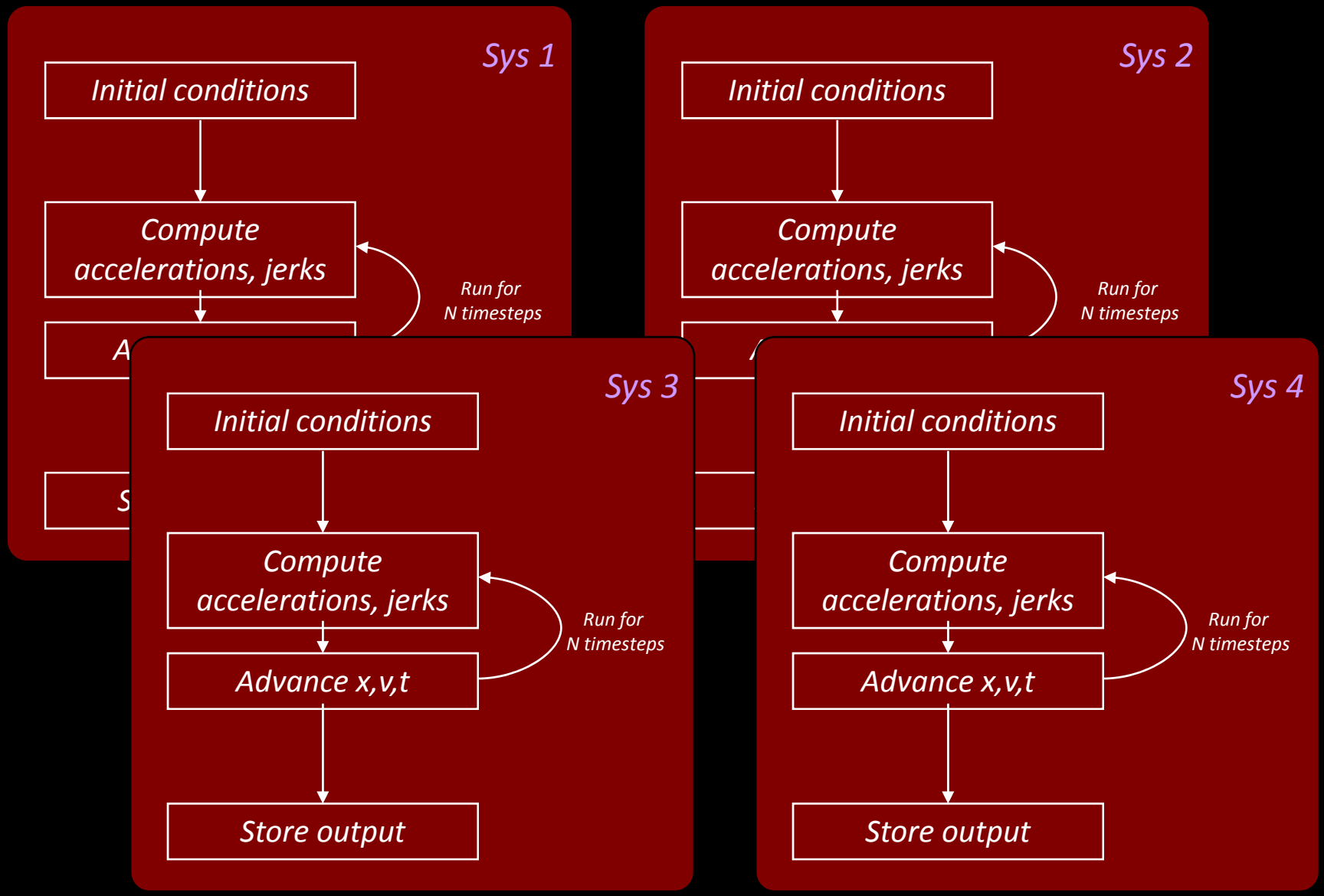
$$\mathbf{j}_i = G\sum_{\substack{j=1 \\ j\neq i}}^{N} M_j \left[\frac{\mathbf{v}_{ji}}{r_{ji}^3} - 3\frac{(\mathbf{r}_{ji}\cdot\mathbf{v}_{ji})\mathbf{r}_{ji}}{r_{ji}^5}\right]$$

Note: the jerk is computed in the same pass as the velocities and positions; the 4th order accuracy improvement it brings comes with a minimal computational cost.

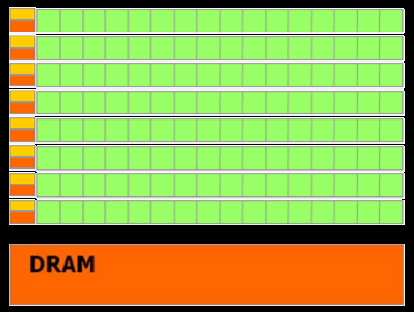
A Simple CPU Implementation



CPU parallelization (trivial: run $N=N_{cores}$ jobs at a time)

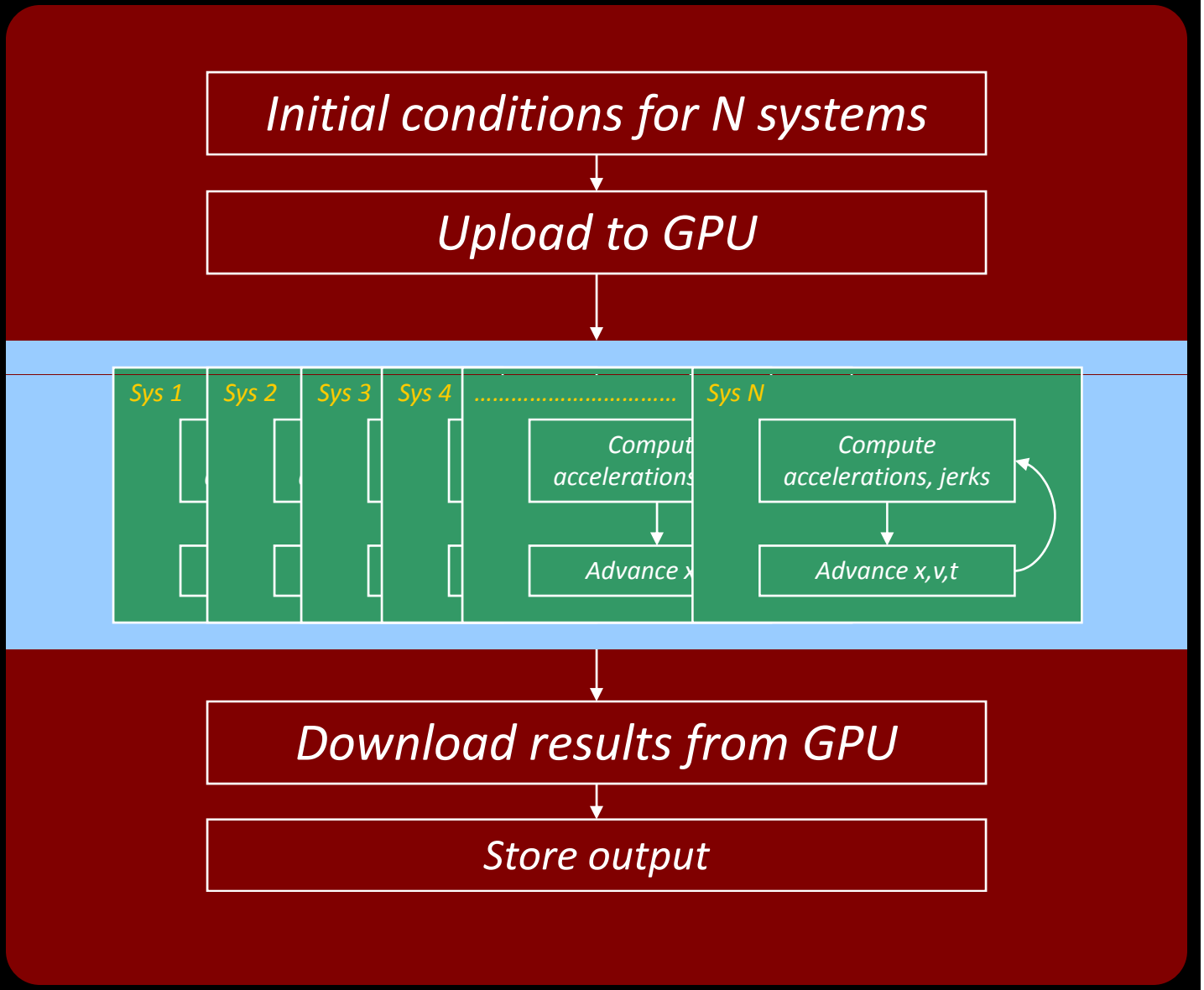


GPU parallelization (slightly nontrivial: *N* jobs in 1 process)

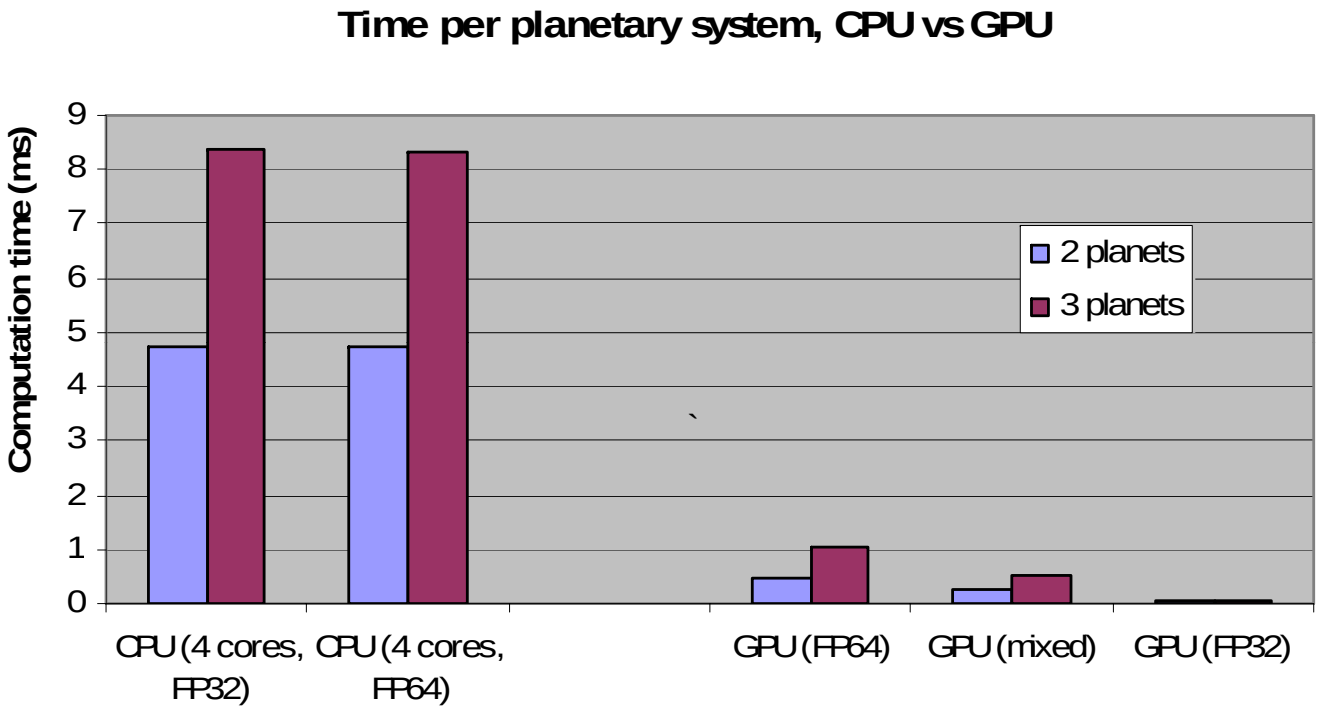


Separate device

All cores run the same program: have to pack N jobs into 1 process



Benchmarks: CPU vs. OpenMP vs. GPU

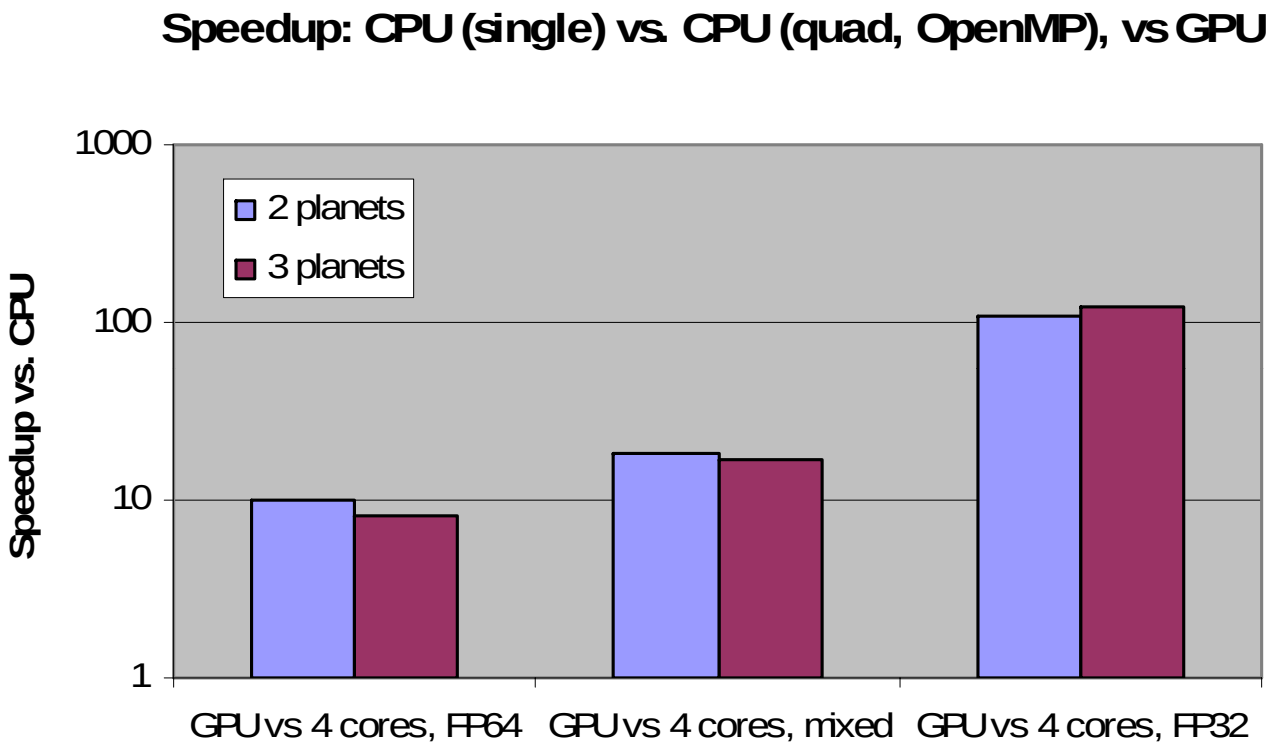


CPU: 2x Dual-Core 2.6 GHz Opteron (total of 4 cores)

GPU: NVIDIA GeForce GTX 280 (240 cores)

Credit: Implementation by Young In Yeo

Benchmarks: CPU vs. OpenMP vs. GPU



***~8-10x** speedup over 4-core OpenMP, DP (!)*

***~110-120x** speedup over 4-core OpenMP, SP (!!)*

Credit: Implementation by Young In Yeo

About parallelization: TMTOWTDI

- Benefits of trivial parallelization
  - Simple, excellent to begin with
  - Virtually the only way of accelerating few-body problems
  - Solves ***our*** problems

- Other ways to do it for larger N (e.g., parallelize force/jerk computation)

$$\mathbf{j}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} M_j \left[\frac{\mathbf{v}_{ji}}{r_{ji}^3} - 3 \frac{(\mathbf{r}_{ji} \cdot \mathbf{v}_{ji}) \mathbf{r}_{ji}}{r_{ji}^5} \right]$$

(Hamada et al. 2007, Portegies-Zwart 2007, …)

Other Results

- Similar speedups for Verlet scheme
- Excellent speedups of random number generation (~30x over 4-core CPU)
- Promising initial benchmarks of Kepler equation solvers (~10x over 4-core CPU)
- Good progress so far

A Moment of Zen…

- "About GPUs, this is NOT…"
- This is about CPUs, 5 years from now
- We must:
  - (Re)learn how to code for 1000-core, shared memory machines
  - Have the basic tools to efficiently use them (e.g., ODE and N-body solvers)
- Obtaining 10-100x speedup NOW doesn't hurt either ☺

Summary

- Highly encouraging initial results (10x-100x speedups on simple, constant-timestep, integrators)
- On track for first release of usable and well documented N-body kernels in Spring 2010.
- Proceeding with development of complex ODE methods (symplectic codes, adaptive timestep, etc.)
- Note: General purpose uses beyond planetary dynamics