6th/8th-order Hermite Integrator & GPU Implementations

University of Tokyo /
National Astronomical Observatory of Japan

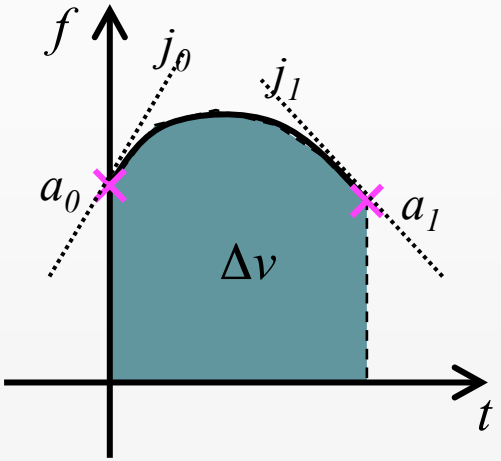
Keigo Nitadori

Abstract

- 6th/8th-order Hermite Integrator
  - Directly calculates up to *snap* (the 2nd derivative of acceleration) or *crackle* (the 3rd derivative).
    - acceleration, jerk, snap, crackle & pop
  - (Of course) used with individual timestep scheme
  - More efficient than the traditional 4th-order one
- GPU Implementations
  - 4th/6th-order NBODY1 implementation with double precision emulation
  - Accelerating NBODY6

Introduction

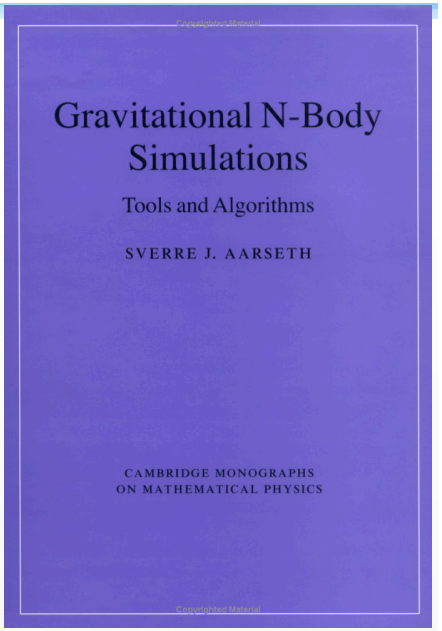
- The 4th order Hermite integrator
  - Directly calculates the acceleration and its time derivative (jerk)
  - Achieves 4th-order without any past information
  - Allows x2 step-size compared to the original Adams integrator



$$\Delta v = \frac{\Delta t}{2}(a_1 + a_0) - \frac{\Delta t^2}{12}(j_1 - j_0)$$

- Further derivatives?
  - How is the calculation cost?
  - Snap requires acceleration, crackle requires jerk.

Force and derivatives
(Aarseth, 2003; Findlay, 1983)



acccel $\boldsymbol{a}_{ij} = m_j \frac{\boldsymbol{r}_{ij}}{r_{ij}^3}$, 38 ops

jerk $\boldsymbol{j}_{ij} = m_j \frac{\boldsymbol{v}_{ij}}{r_{ij}^3} - 3\alpha \boldsymbol{a}_{ij}$, } 60 ops

snap $\boldsymbol{s}_{ij} = m_j \frac{\boldsymbol{a}_j - \boldsymbol{a}_i}{r_{ij}^3} - 6\alpha \boldsymbol{j}_{ij} - 3\beta \boldsymbol{a}_{ij}$, } 97 ops

crackle $\boldsymbol{c}_{ij} = m_j \frac{\boldsymbol{j}_j - \boldsymbol{j}_i}{r_{ij}^3} - 9\alpha \boldsymbol{s}_{ij} - 9\beta \boldsymbol{j}_{ij} - 3\gamma \boldsymbol{a}_{ij}$, } 144 ops

where

$$\alpha = \frac{\boldsymbol{r}_{ij} \cdot \boldsymbol{v}_{ij}}{r_{ij}^2},$$

$$\beta = \frac{|\boldsymbol{v}_{ij}|^2 + \boldsymbol{r}_{ij} \cdot (\boldsymbol{a}_j - \boldsymbol{a}_i)}{r_{ij}^2} + \alpha^2,$$

$$\gamma = \frac{3\boldsymbol{v}_{ij} \cdot (\boldsymbol{a}_j - \boldsymbol{a}_i) + \mathrm{r}_{ij} \cdot (\boldsymbol{j}_j - \boldsymbol{j}_i)}{r_{ij}^2} + \alpha(3\beta - 4\alpha^2),$$

and

$$\boldsymbol{r}_{ij} = \boldsymbol{r}_j - \boldsymbol{r}_i,$$

$$\boldsymbol{v}_{ij} = \boldsymbol{v}_j - \boldsymbol{v}_i.$$

- ✓~50% increase per derivative
- ✓not so expensive as expected

Predictor & Corrector

for the 6th-order integrator

Predictor

$$\begin{aligned}
\boldsymbol{r}_{i,p} &= \boldsymbol{r}_i + \boldsymbol{v}_i\Delta t + \frac{1}{2}\boldsymbol{a}_i\Delta t^2 + \frac{1}{6}\boldsymbol{j}_i\Delta t^3 + \frac{1}{24}\boldsymbol{s}_i\Delta t^4 + \frac{1}{120}\boldsymbol{c}_i\Delta t^5, \\
\boldsymbol{v}_{i,p} &= \boldsymbol{v}_i + \boldsymbol{a}_i\Delta t + \frac{1}{2}\boldsymbol{j}_i\Delta t^2 + \frac{1}{6}\boldsymbol{s}_i\Delta t^3 + \frac{1}{24}\boldsymbol{c}_i\Delta t^4, \\
\boldsymbol{a}_{i,p} &= \boldsymbol{a}_i + \boldsymbol{j}_i\Delta t + \frac{1}{2}\boldsymbol{s}_i\Delta t^2 + \frac{1}{6}\boldsymbol{c}_i\Delta t^3.
\end{aligned}$$

Acceleration is also predicted

Corrector

$$\begin{aligned}
\boldsymbol{v}_{i,c} &= \boldsymbol{v}_{i,0} + \frac{\Delta t}{2}(\boldsymbol{a}_{i,1} + \boldsymbol{a}_{i,0}) - \frac{\Delta t^2}{10}(\boldsymbol{j}_{i,1} - \boldsymbol{j}_{i,0}) + \frac{\Delta t^3}{120}(\boldsymbol{s}_{i,1} + \boldsymbol{s}_{i,0}), \\
\boldsymbol{r}_{i,c} &= \boldsymbol{r}_{i,0} + \frac{\Delta t}{2}(\boldsymbol{v}_{i,c} + \boldsymbol{v}_{i,0}) - \frac{\Delta t^2}{10}(\boldsymbol{a}_{i,1} - \boldsymbol{a}_{i,0}) + \frac{\Delta t^3}{120}(\boldsymbol{j}_{i,1} + \boldsymbol{j}_{i,0}).
\end{aligned}$$

Timestep criterion

Original Aarseth criterion for 4th order integrators

$$\Delta t = \sqrt{\eta \frac{|\boldsymbol{a}||\boldsymbol{a}^{(2)}| + |\boldsymbol{a}^{(1)}|^2}{|\boldsymbol{a}^{(1)}||\boldsymbol{a}^{(3)}| + |\boldsymbol{a}^{(2)}|^2}}, \quad \text{where} \quad \boldsymbol{a}^{(k)} \equiv \frac{d^k \boldsymbol{a}}{dt^k},$$

- (only) works well with 4th order schemes

Generalized Aarseth criterion

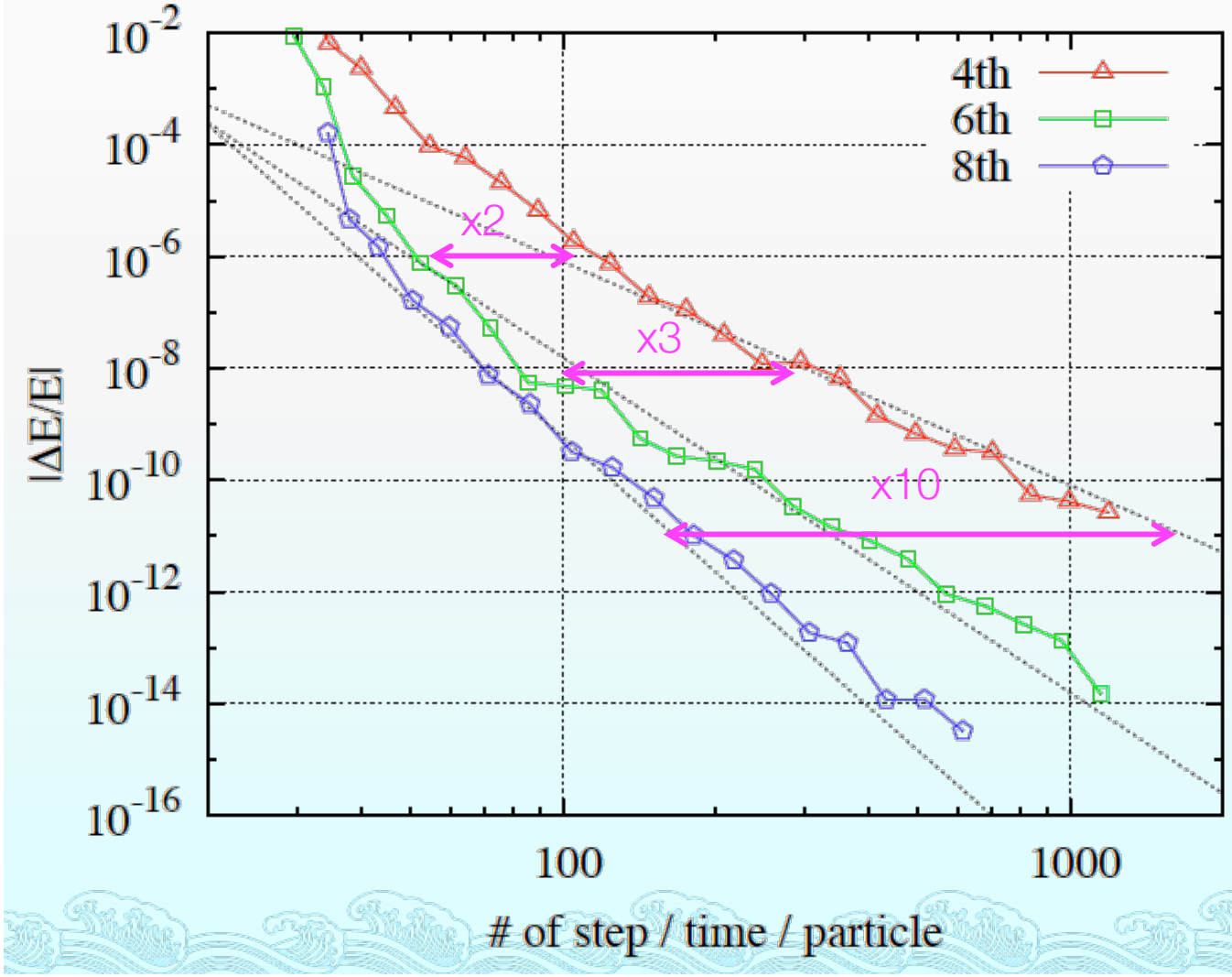
$$\Delta t = \eta \left(\frac{A^{(1)}}{A^{(p-2)}} \right)^{1/(p-3)}, \quad \text{where} \quad A^{(k)} \equiv \sqrt{|\boldsymbol{a}^{(k-1)}||\boldsymbol{a}^{(k+1)}| + |\boldsymbol{a}^{(k)}|^2},$$

- reflects the highest derivative $\boldsymbol{a}^{(p-1)}$
- works well with pth order schemes

Implementation

- Fully modularized with C++
- Integrator dependent parts are defined in *header* files
  - Particle class, Predictor class, Force class
  - Predictor, corrector, timestep criterion
  - ```
    #include “hermite4.h”
    // #include “hermite6.h”
    // #include “hermite8.h”
    ```
- Others (file I/O, blockstep algorithms) are common files
    ```

Error for step-count



$N = 1024$

$\varepsilon = 4/N = 1/256$

Heggie unit

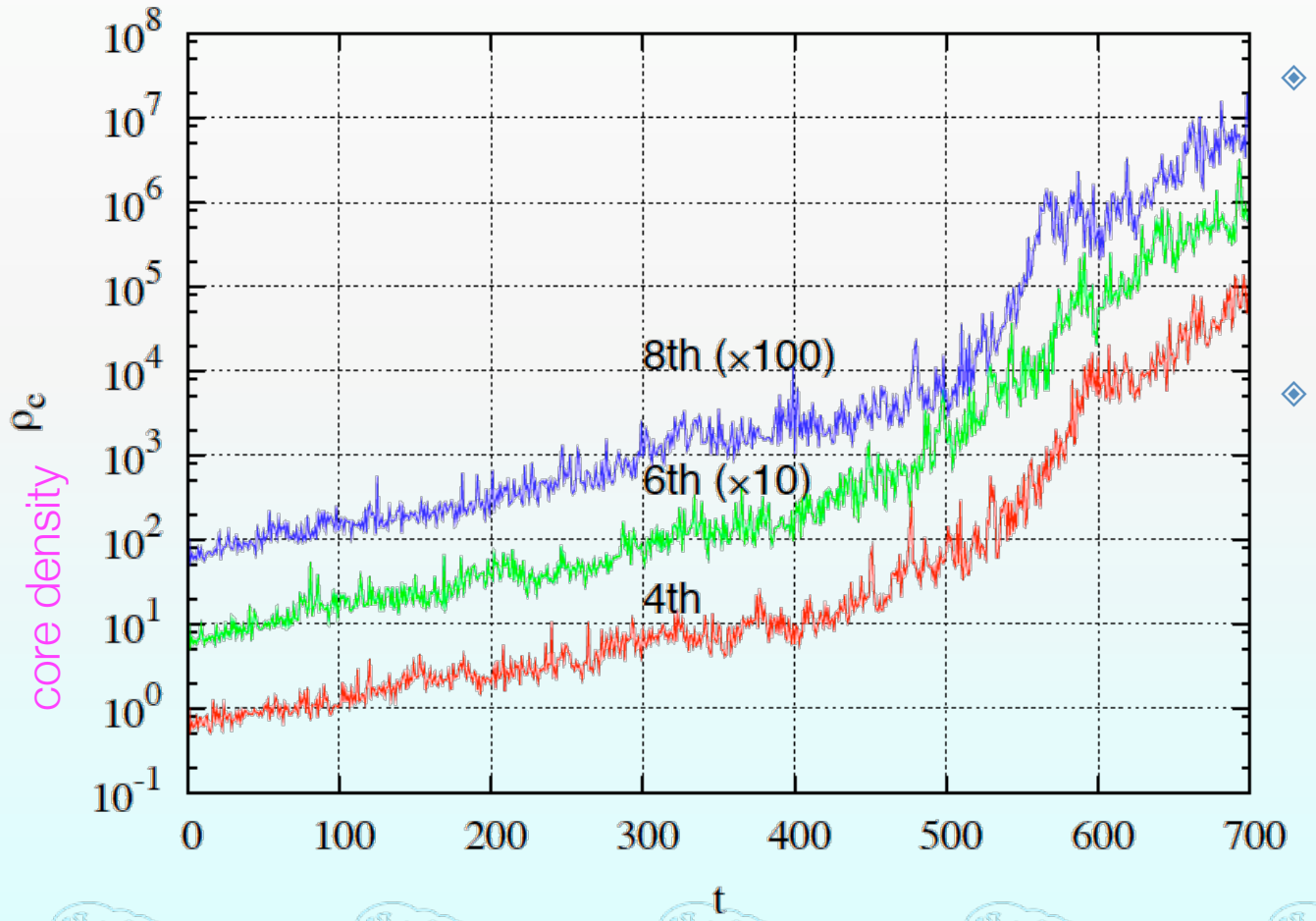
Plummer model

$0.125 < t < 10.125$

individual (block) timestep algorithm

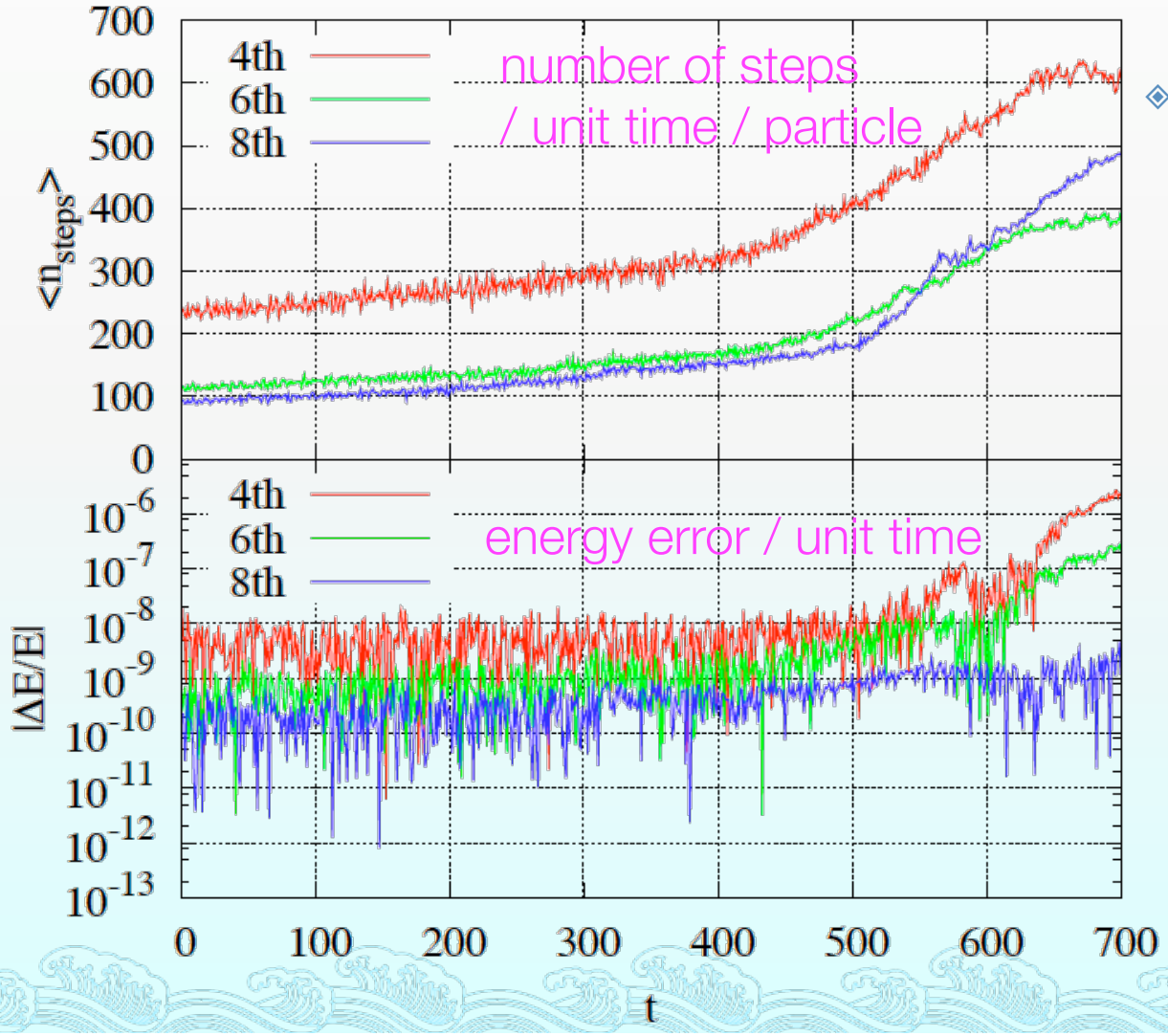
Core collapse test

$N = 1024$
$\varepsilon = 4/N = 1/256$
Heggie unit
Plummer model



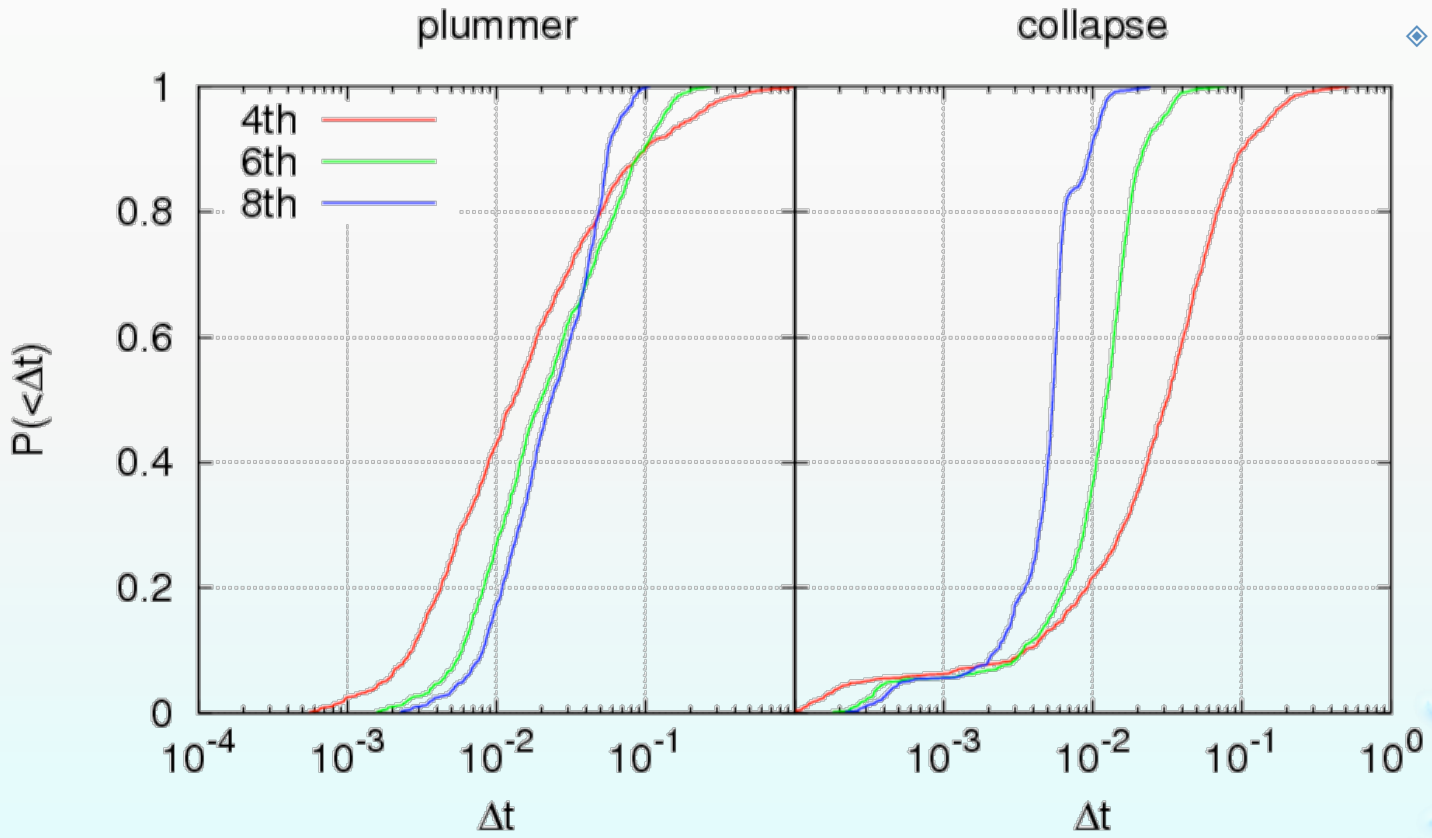
- Good agreement among 4th, 6th and 8th order integrators
- Much difference in step-count and error (next slide)

Energy error for step count



- The 8th-order integrator:
  - Step-count increases by x5
  - Error is kept small

(Cumulative) timestep distribution



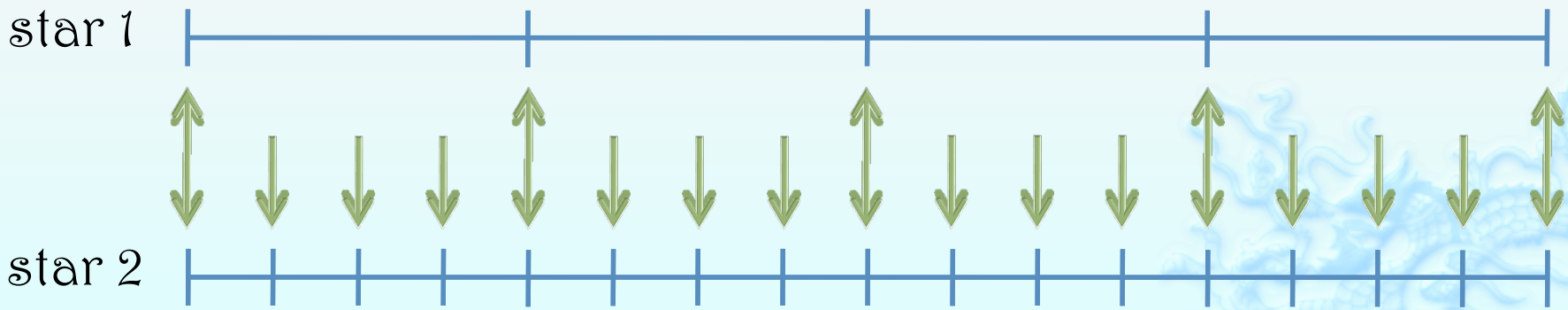
- In high order schemes (reflects high order derivatives),Time step has an upper limit after the core collapse

What's happened?

- In the 8th-order scheme
  - Timesteps of outer region stars were shrunk by a *high-frequency force variation* from the core (timestep criterion reflects high order derivatives)
- In the 4th-order scheme
  - Timesteps were not shrunk, but high-frequency caused a large error (failed to integrate correctly)

Is individual timestep scheme correct?

- Breaks force symmetric law
- (If the timestep ratio is large [$\Delta t_1 > \sim T_2$]) star 1 cannot integrate the force from star 2 correctly.



Summary of part 1

- For typical use ($\Delta E \sim 10^{-8}$), 6th-order Hermite integrator seems to be optimal
- The individual timestep scheme might not be correct
  - Cannot treat widely spread timescale!?
  - Higher order integrators tend to be shared timestep

Part2: GPU *N*-body

- GPU is originally for 3D games, but now very hot among *N*-body programmers
- 450 Gflops single precision peak is available in $300
- Performance in single precision N^2 shared step is very good (>300 Gflops)

Some recent works

- > 100 Gflops
  - Hamada & Iitaka (2007) (CUDA) (Chamomile)
  - Belleman et al. (2007) (CUDA) (`kirin`)
  - Elsen et al. (2007) (ATI X1900XTX)
  - Nyaland et al. (2007) (CUDA)
  - Schive et al. (2007) (CUDA)
- All of them are single precision result
- Except for `kirin`, N^2 shared timestep

What I did

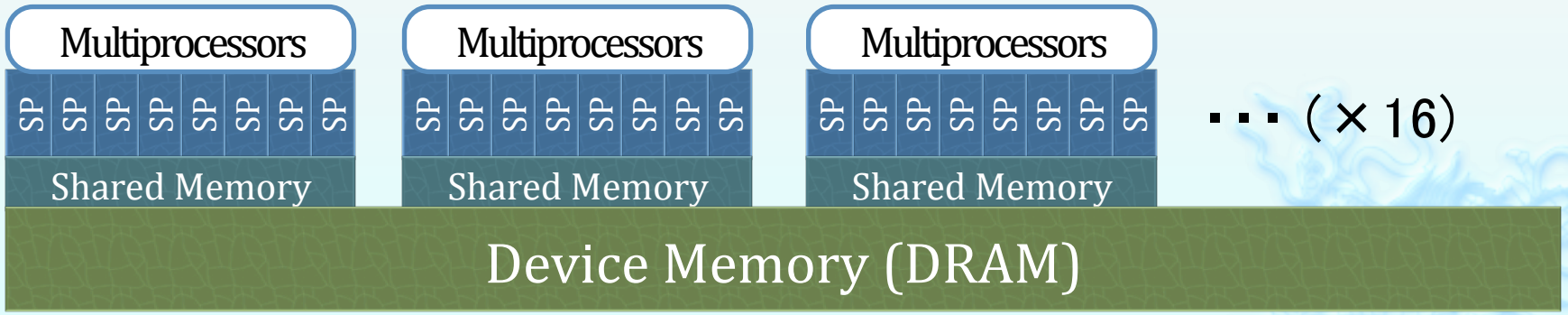
- Implemented of 4th/6th-order HITS (Hermite Individual Timestep Scheme) on GPU with CUDA
- Emulated of double precision in some parts of calculation
  - Got a practical accuracy for collisional simulations
- Achieved 200 Gflops (N=64k, 6th-order), as fast as Teraflops GRAPE-6

What I did

- Implemented of 4th/6th-order HITS (Hermite Individual Timestep Scheme) on GPU with CUDA
- Emulated of double precision in some parts of calculation
  - Got a practical accuracy for collisional simulations
- Achieved 200 Gflops (N=64k, 6th-order), as fast as Teraflops GRAPE-6

Architecture of GPU (G80/G92)

- 2 levels of parallel processors
  - 16 multiprocessors which share the main memory
    - MIMD (Multiple Instructions Multiple Data)
  - 8 stream processors with 16 kB shared memory
    - Instruction is shared, code path must be the same
    - SIMD (Single Instruction Multiple Data)



Programming model of CUDA

- Apparently, SPMD (Single Program Multiple Data)
  - There are multiple blocks and multiple threads
  - Blocks can execute different code paths
  - Threads must share the code path

```
__global__ void GPU_kernel(float *mem, ...){
    int bid = blockIdx.x;
    int bsize = gridDim.x;
    int tid = threadIdx.x;
    int tsize = blockDim.x;
    ...
}
void host_func(...){
    dim3 blocks (32, 1, 1);
    dim3 threads (128, 1, 1);
    cudaMemcpy(mem_dev, mem_host, cudaMemcpyHostToDevice);
    GPU_kernel <<<blocks, threads>>> (mem_dev, ...);
    cudaMemcpy(mem_host, mem_dev, cudaMemcpyDeviceToHost);
}
```

Where double precision is needed

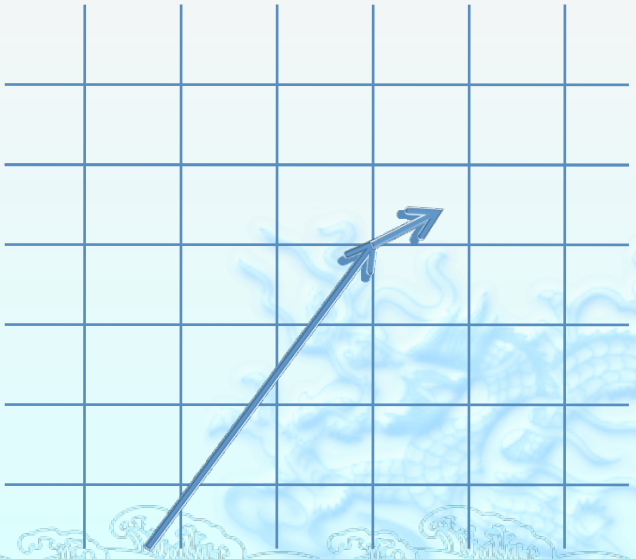
```
struct vec3{ float x, y, z; ...};
vec3 pos[N], acc[N];
float mass[N], pot[N];
for(int i=0; i<N; i++){
    vec3  ai(0);
    float pi = 0;
    for(int j=0; j<N; j++){
        if(j==i) continue;
        vec3  dr = pos[j] - pos[i];
        float r2 = dr * dr;
        float r2inv = 1./r2;
        float rinv = sqrt(r2inv);
        rinv *= mass[j];
        float r3inv = rinv * r2inv;
        pi -= rinv;
        ai += r3inv * dr;
    }
    pot[i] = pi;
    acc[i] = a[i];
}
```

- Differential of Positions
  - A close encounter causes a digit loss
- Accumulation of force
  - summing up many small numbers
- Most of intermediate calculations are performed in single precision
  - HARP, GRAPE-2/4/6

Double precision emulation I: Coordinate

```
struct Float2{
    float high, low;
    //  construct from a DP word
    __host__
    Float2(double x){
        x *= (1<<20);
        high = int(x);   // integer part
        low  = x - high;// fraction part
        high /= (1<<20);
        low  /= (1<<20);
    }
    // differential, 3 ops
    __device__
    float operator - (Float2 rhs){
        return (high - rhs.high)
             + (low  - rhs.low );
    }
};
```

- Split a DP word to 2 SP words
- Coordinate of a grid point and relative coordinate
- 3 operations



Double precision emulation II: Accumulator

```
struct Float2{
    float high, low;
    // accumulate, 4 ops
    __device__
    void operator += (float a){
        float tmp = high;
        high += a;
        // detect the round-off error
        float da = a - (high - tmp);
        low += da
    }
    __host__
    operator double (){
        return double(high)
             + double(low);
    }
};
```

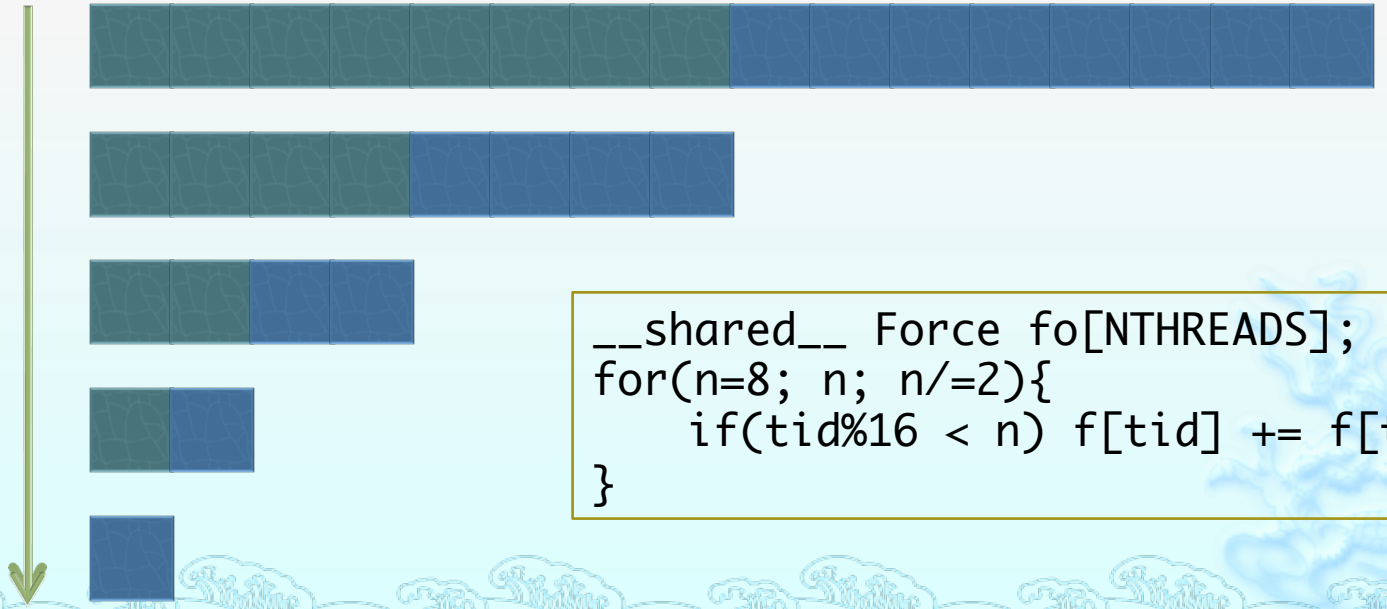
- Detect the round-off error and accumulate again
- 4 operations
- Overhead per interaction: (2+3)*3=15 ops

Predictor

- Predictor is also performed in GPU
  - The next most expensive to the force calculation
    - Force: $O(NN_{\mathrm{act}})$, Predictor: $O(N)$, Corrector, etc: $O(N_{\mathrm{act}})$
  - Minimize the communication overhead
- Position predictor is added to the lower word of the position
  - $x_{\mathrm{H,p}} \leftarrow x_{\mathrm{H}}$
    $x_{\mathrm{L,p}} \leftarrow x_{\mathrm{L}} + \Delta x$
    $v_{\mathrm{p}} \leftarrow v + \Delta v$

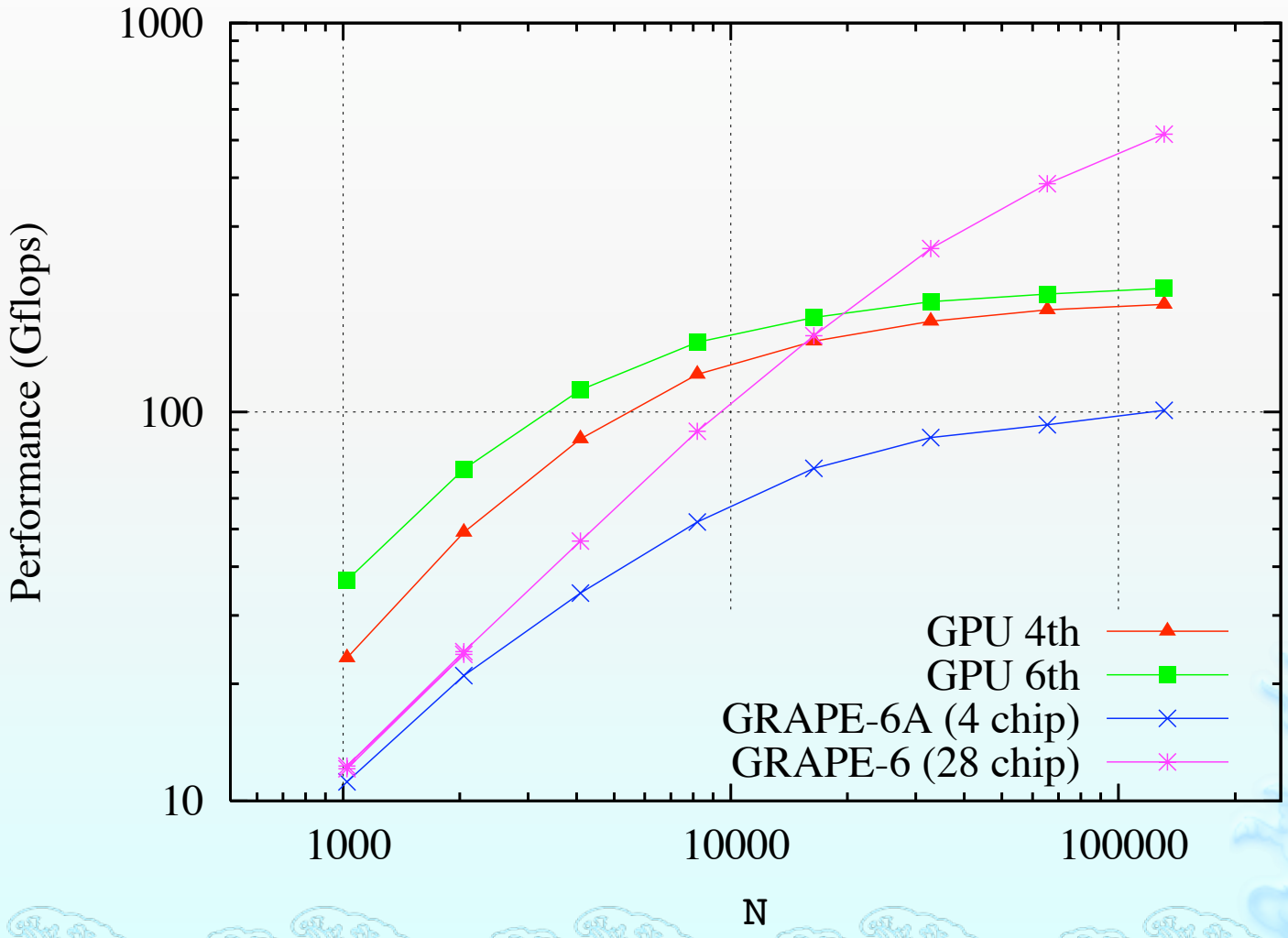
Force reduction

- We need a force reduction kernel separateley
  - otherwise, large communication overhead
- 16 blocks calculates partial force (j-parallel)
  - Blocks write 16 forces to the DRAM
  - Reduction kernel reduces 16 forces to 1



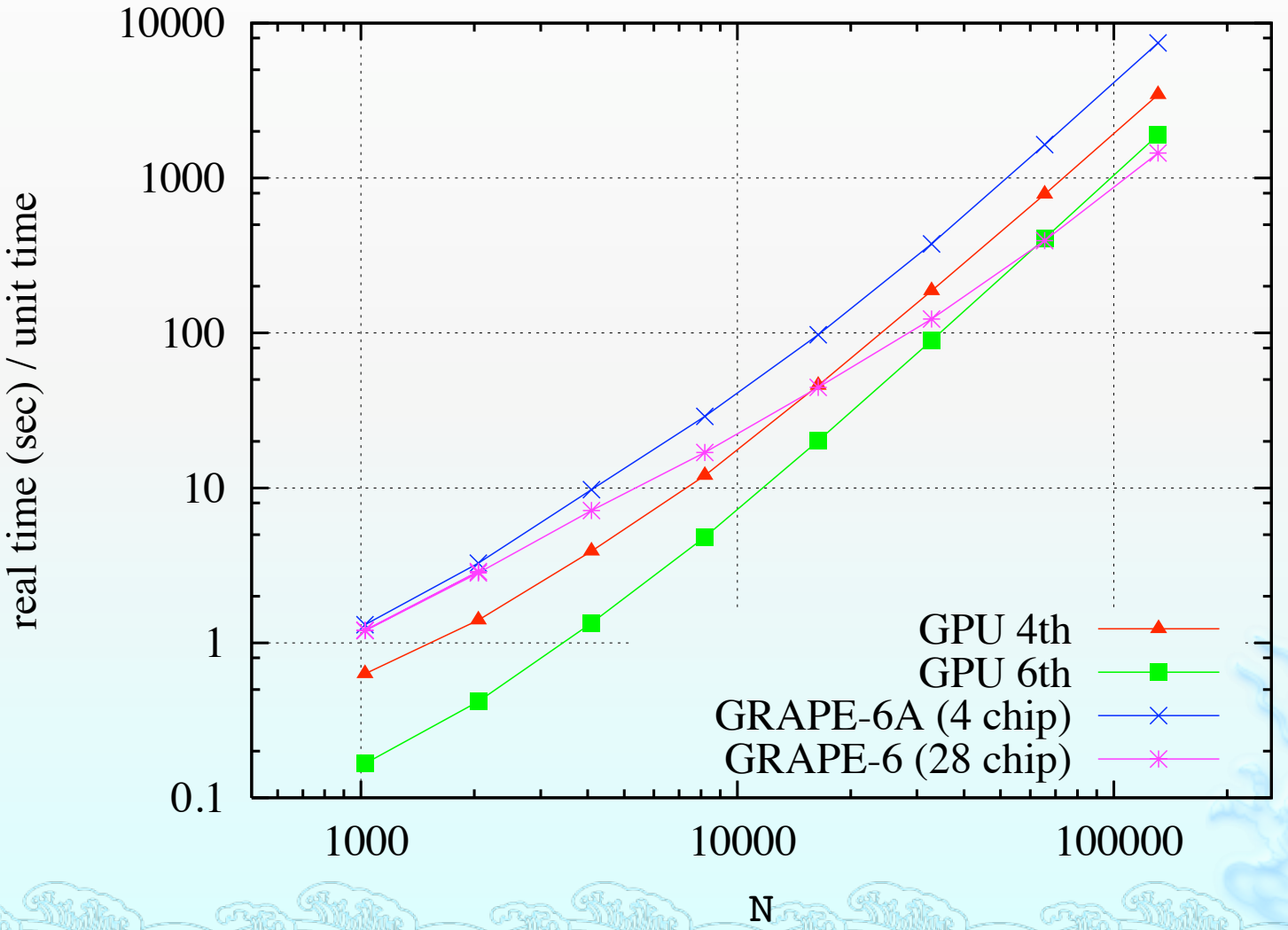
```
__shared__ Force fo[NTHREADS];
for(n=8; n; n/=2){
    if(tid%16 < n) f[tid] += f[tid+n];
}
```

Performance (Gflops)



- $\varepsilon = 4/N$
- x2 better than GRAPE-6A
- Small N performance is good in GPU

Wall-clock-time



- $\eta_{4th}=0.1$
- $\eta_{6th}=0.75$
- 6th-order gains x2 to the 4th
- GPU is as fast as Teraflops GRAPE-6 at N=64k

Summary of Part2

- GRAPE-6 could be replaced with GPU
- How about GRAPE-DR
  - 1 chip: 600,000 JPY, as fast as GPU
  - 4 chips: 1,500,000 JPY, 4x faster in peak
- For both GPU and GRAPE-DR, 6th-order Hermite integrator would perform better
  - That's why I'm not interested in writing a GRAPE-6 compatible library

Part3: NBODY6/GPU

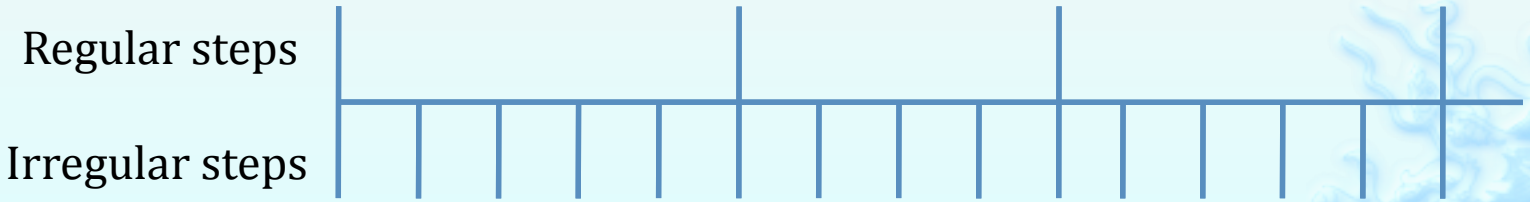
- NBODYx series by Sverre Aarseth:
  - A collection of *N*-body codes with a long history

	Integrator	Singularity	Neighbour scheme	GRAPE
NBODY1	Adams	Softening	No	(No)
NBODY2	Adams	Softening	Yes	No
NBODY3	Adams	MREG	No	No
NBODY4	Hermite	MREG	No	Yes
NBODY5	Adams	MREG	Yes	No
NBODY6	Hermite	MREG	Yes	No

Others: NBODY1h, NBODY2h, NBODY0, NBODY7

Ahmad-Cohen neighbour scheme

- An efficient integration method for *N*-body calculation
- Less frequently evaluate the total force and more frequently evaluate the force from the neighbours
  - Regular step: Calculate full interactions and build the neigubour list
  - Irregular step: Calculate only the force from neighbour particles



Previously

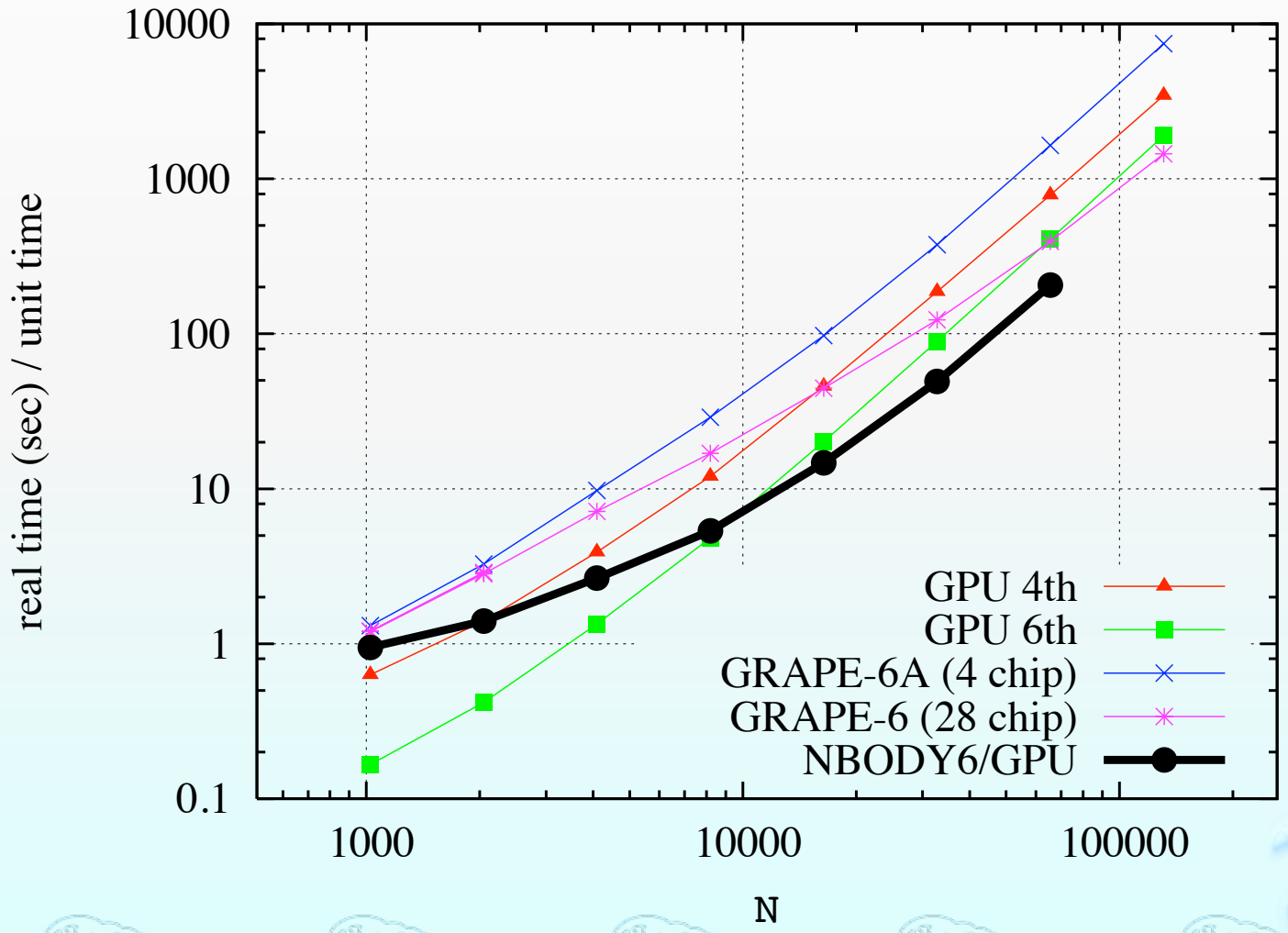
- NBODY4 with brute force has been accelerated by special purpose hardware
- NBODY6 has been an efficient code, but has not been accelerated

So, I and Sverre tried to accelerate NBODY6 with GPU

Implementation

- GPU calculates only the regular step which has dominated 95% in the profiler
  - Calculates force from distant particles and make neighbour list
  - Irregular step is performed on CPU with double precision
- Full single precision calculation was OK
  - Prediction or reduction is performed on the host

Performance



- $\eta_{reg} = 0.03$
- $\eta_{irr} = 0.02$

- Excellent

Summary of Part 3

- We can start a simulation of 50,000 body cluster from 100,000 JPY
  - Previously 5,000,000 JPY (Teraflops GRAPE-6)
- Future works:
  - Multiple GPU
    - Quad GPU environment with dual 9800GX2
  - 6th-Order NBODY6