

# N-Body Problems

Parallel programming  
Week 9

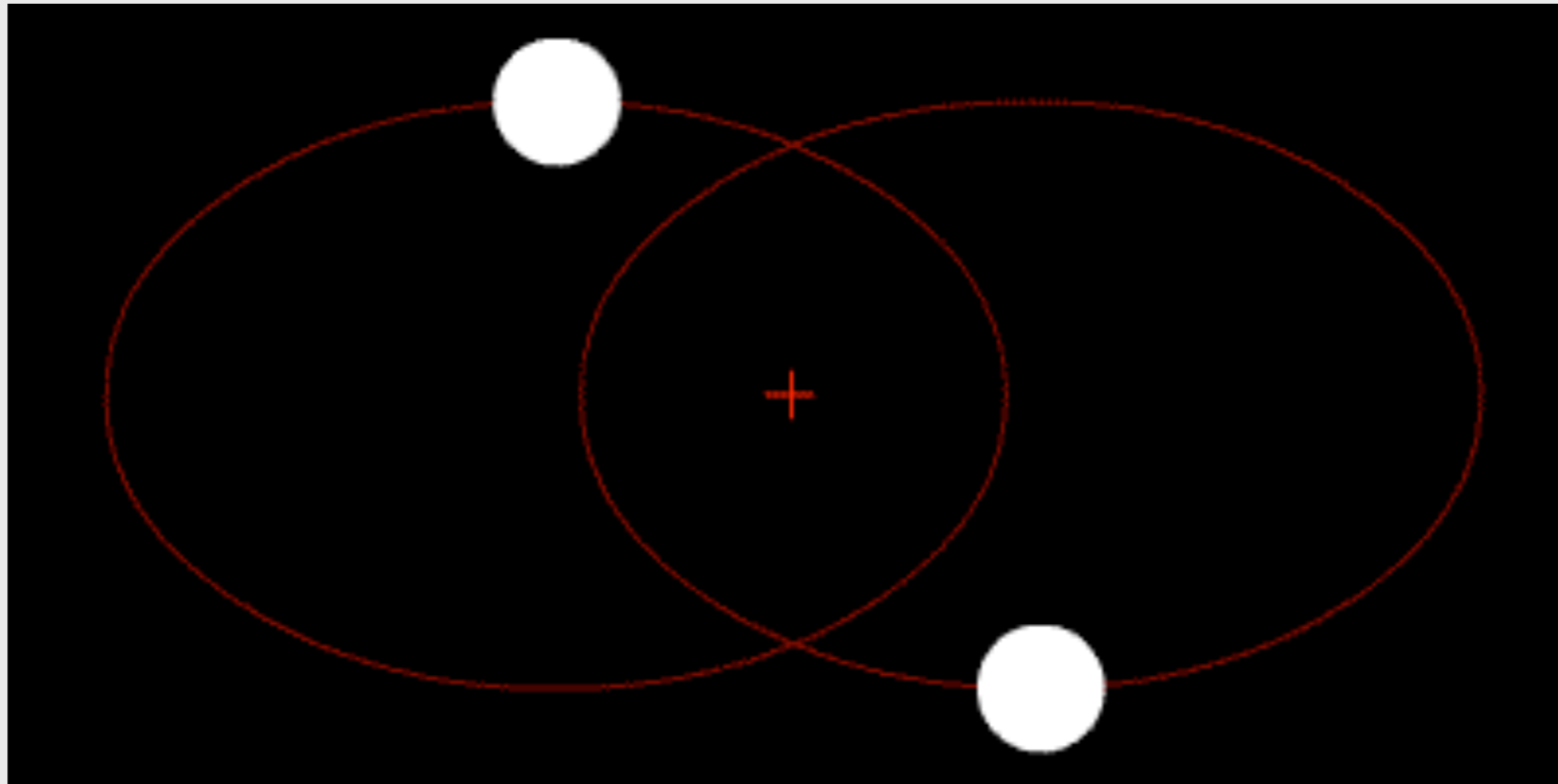
# N-body problems

- typical parallelizable problem
- $n$  bodies with pair-wise interaction
- typical example: astronomy
- bodies approximated as point masses  
(proved by Newton)

# Equation system

$$m_j \ddot{\mathbf{q}}_j = \gamma \sum_{k \neq j} \frac{m_j m_k (\mathbf{q}_k - \mathbf{q}_j)}{|\mathbf{q}_k - \mathbf{q}_j|^3}, j = 1, \dots, n$$

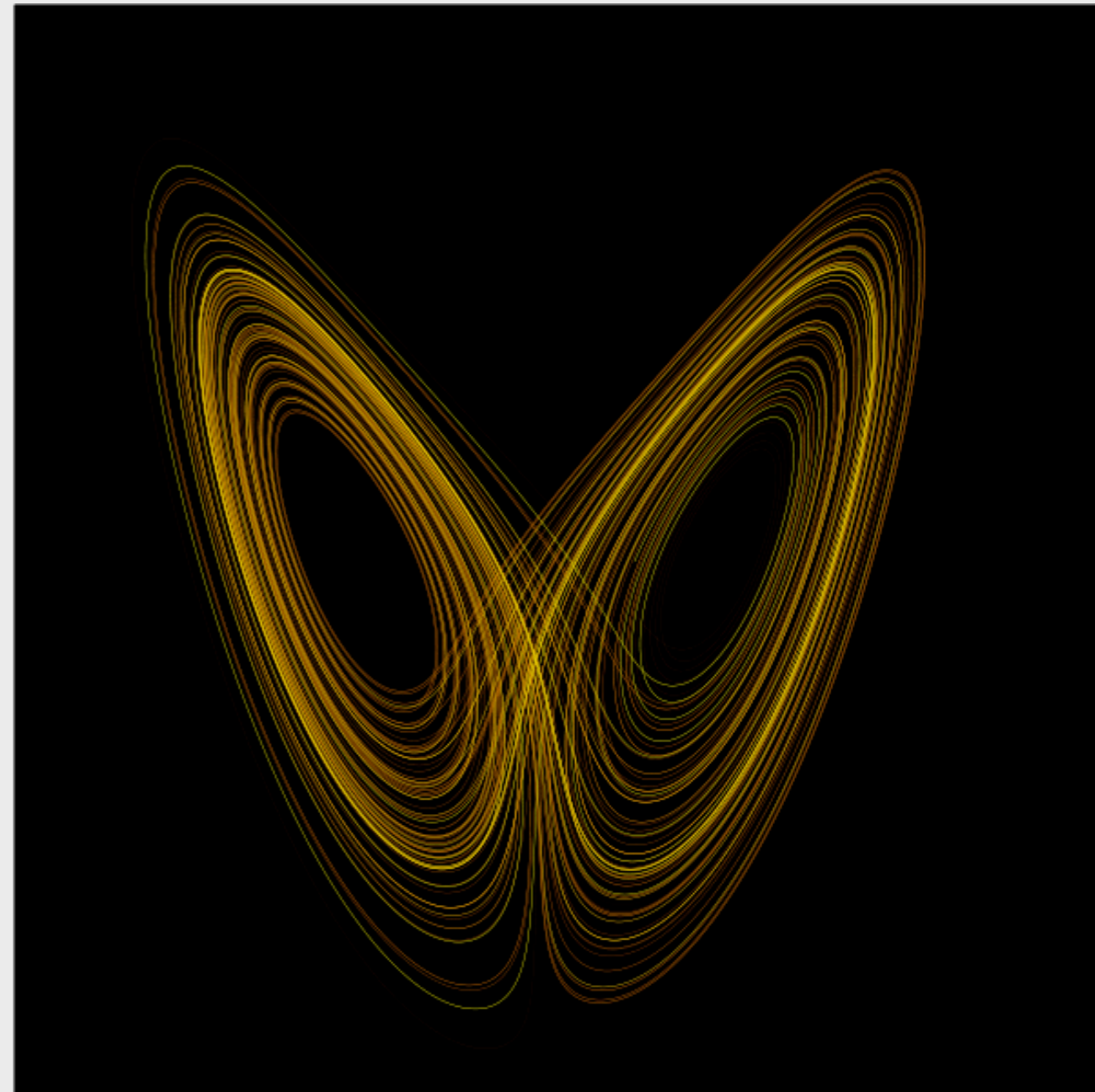
# $N=2$

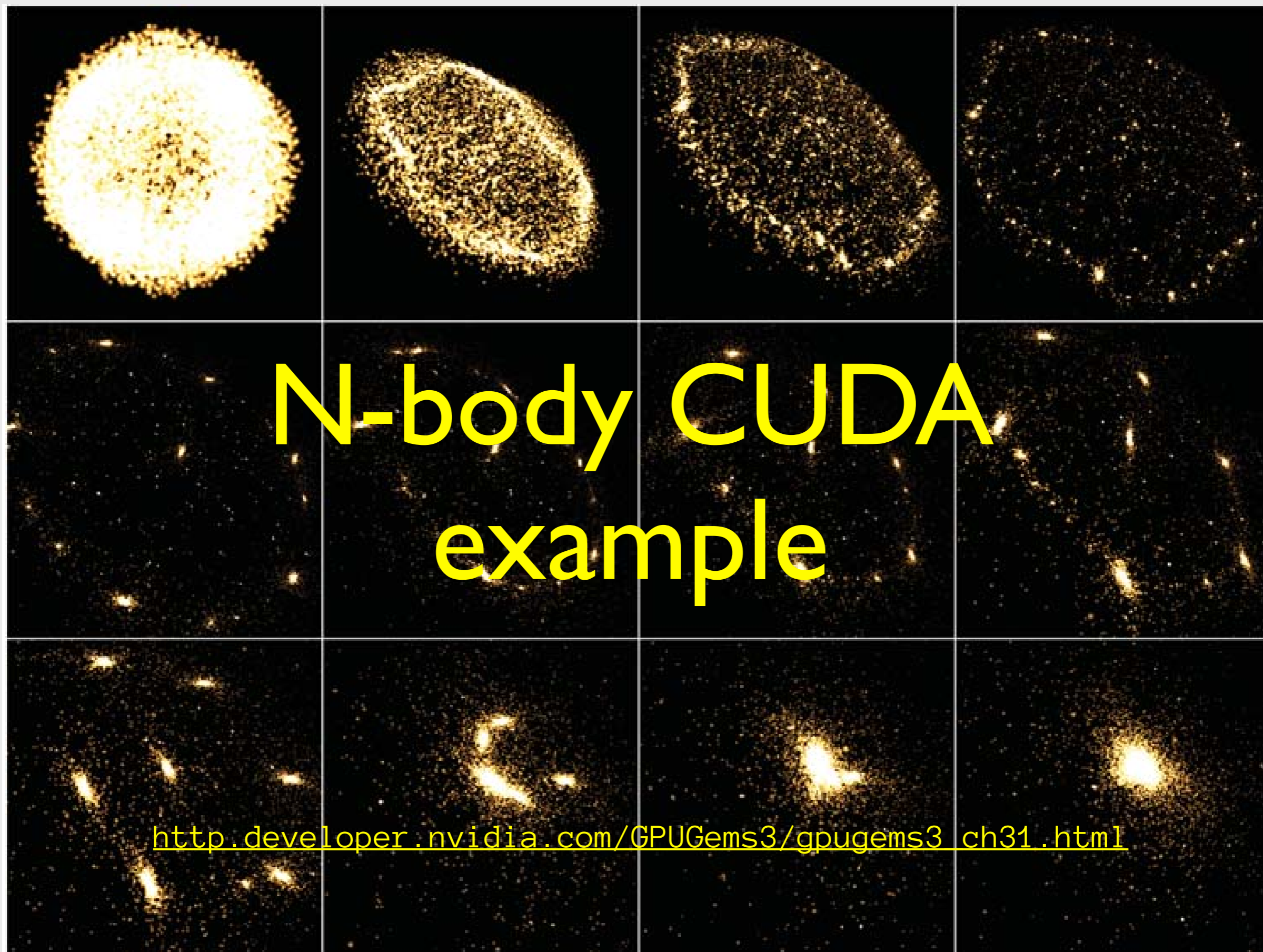


Can be resolved *analytically*.

# $N \geq 2$

- No analytic solution!
- Must use *numeric* solution (or simulation)
- *Chaotic* behaviour
- Singularities
  - collisions
  - divergence
  - [CUDA](#)





# N-body CUDA example

[http://developer.nvidia.com/GPUGems3/gpugems3\\_ch31.html](http://developer.nvidia.com/GPUGems3/gpugems3_ch31.html)

- all-pairs method
- “brute force”  $O(N^2)$  complexity
- typically used for close-range interaction
  - far-field interactions approximated
  - valid only for well separated systems
- focus on the all-pairs,  $N=16,384$ 
  - $N^2 = 268,435,456$

- 10 B interactions/s
- 200 GFlops (close to peak performance)
- hallmark of a well-optimised algorithm

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|},$$

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}.$$

# Singularities

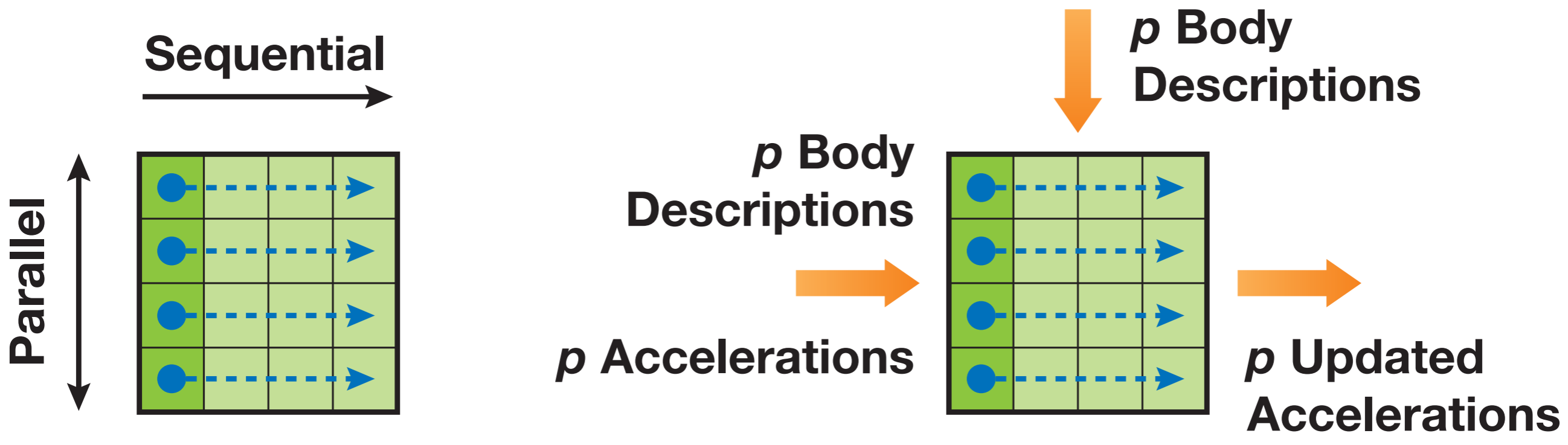
$$\mathbf{F}_i \approx Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left( \|\mathbf{r}_{ij}\|^2 + \varepsilon^2 \right)^{3/2}}.$$

Softening factor  
("galaxies")

$$\mathbf{a}_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left( \|\mathbf{r}_{ij}\|^2 + \varepsilon^2 \right)^{3/2}}.$$

# Managing memory

- $N \times N$  available parallelism but
  - $N \times N$  grid requires too much memory
- serialise some computations
- use *tiles* of size  $p \times p$ 
  - $2p$  body descriptions required
  - $p$  bodies can be reused for next tile
  - total effect updated, stored in shared memory



```
__device__ float3
```

```
bodyBodyInteraction(float4 bi, float4 bj, float3 ai)
```

```
{
```

```
float3 r;
```

for coalesced access

save memory

```
// r_ij
```

```
r.x = bj.x - bi.x;
```

```
r.y = bj.y - bi.y;
```

```
r.z = bj.z - bi.z;
```

```
// distSqr = dot(r_ij, r_ij) + EPS^2 [6 FLOPS]
```

```
float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;
```

```
// invDistCube = 1/distSqr^(3/2) [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
```

```
float distSixth = distSqr * distSqr * distSqr;
```

```
float invDistCube = 1.0f/sqrtf(distSixth);
```

```
// s = m_j * invDistCube [1 FLOP]
```

```
float s = bj.w * invDistCube;
```

```
// a_i = a_i + s * r_ij [6 FLOPS]
```

```
ai.x += r.x * s;
```

```
ai.y += r.y * s;
```

```
ai.z += r.z * s;
```

20 FPO

```
return ai;
```

```
}
```

# Tile calculation

(one object interacting with  $p$  objects)

```
__device__ float3  
tile_calculation(float4 myPosition, float3 accel)  
{  
    int i;  
    extern __shared__ float4[] shPosition;  
    for (i = 0; i < blockDim.x; i++) {  
        accel = bodyBodyInteraction(myPosition, shPosition[i], accel);  
    }  
    return accel;  
}
```

$p$

no bank  
conflicts

actually called computeBodyAccel(float4 bodyPos,  
float4\* positions, int numBodies) in the code!

# Thread blocks and accelerations in dev memory

```

__global__ void
calculate_forces(void *devX, void *devA)
{
    extern __shared__ float4[] shPosition;

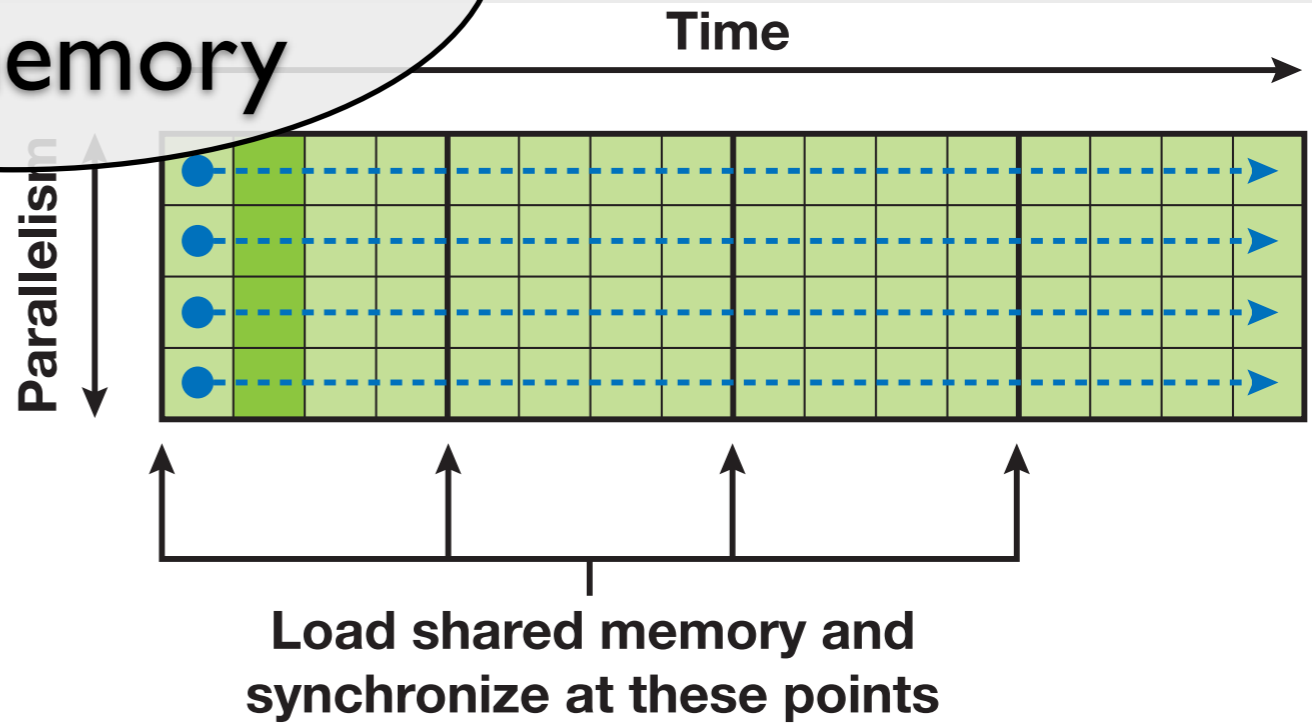
    float4 *globalX = (float4 *)devX;
    float4 *globalA = (float4 *)devA;
    float4 myPosition;
    int i, tile;
    float3 acc = {0.0f, 0.0f, 0.0f};
    int gtid = blockIdx.x * blockDim.x + threadIdx.x;

    myPosition = globalX[gtid];

    for (i = 0, tile = 0; i < blockDim.x; i += blockDim.x, tile++)
    {
        int idx = tile * blockDim.x + threadIdx.x;
        shPosition[threadIdx.x] = globalX[idx];
        __syncthreads();
        acc = tile_calculation(myPosition, acc);
        __syncthreads();
    }

    // Save the result in global memory for the integration step.
    float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
    globalA[gtid] = acc4;
}

```



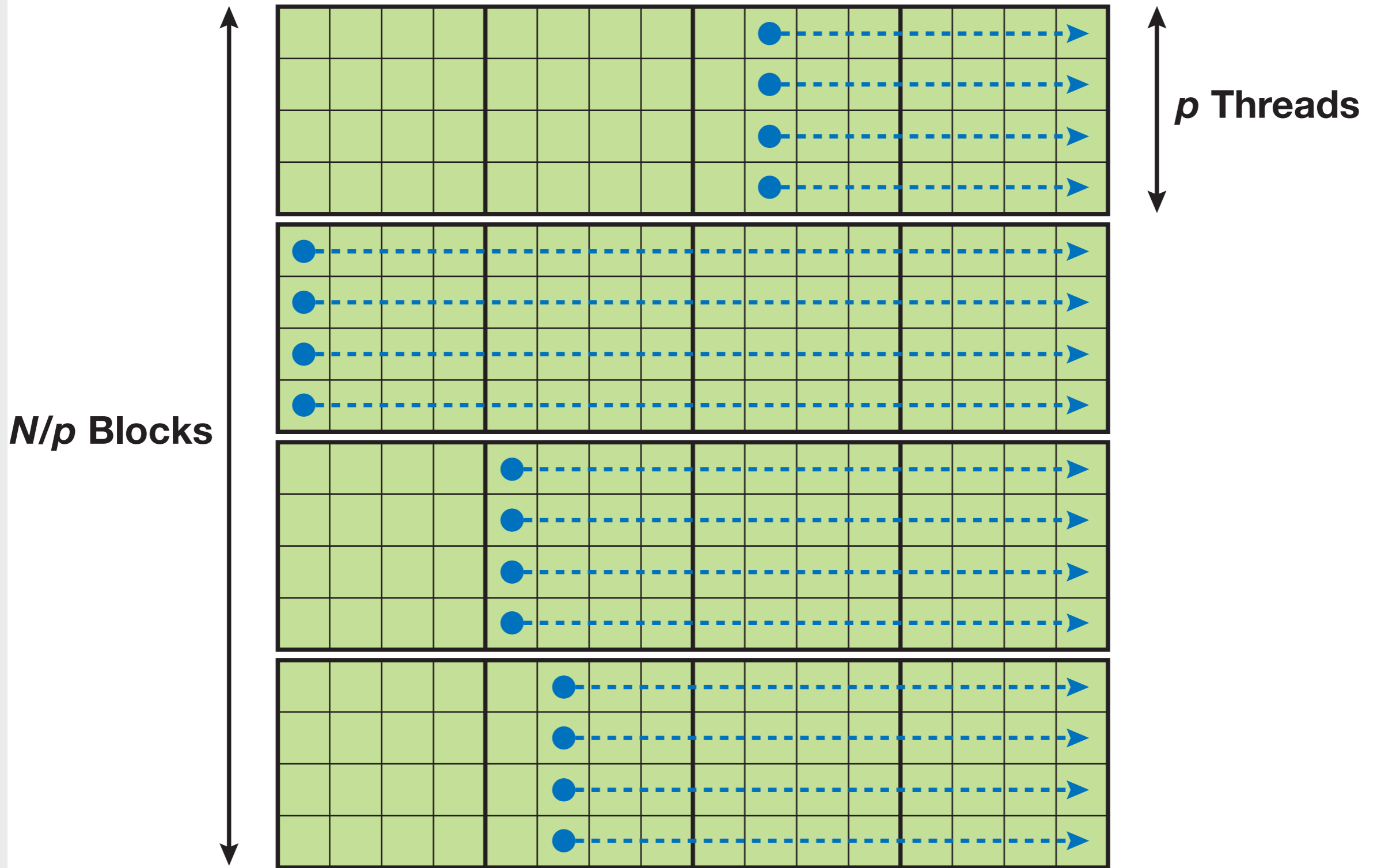
ensure all local memory is loaded

ensure all threads are finished

# The Grid

- each kernel has  $p$  objects interacting with  $N$
- $p$  threads per block, one thread per object
- we need a 1D grid of size  $N/p$
- $N$  threads,  $N$  interactions each

$N$  Bodies



$N/p$  Blocks

$p$  Threads

$p$  steps between loads from global memory

# Integrating the bodies

```
__global__ void
integrateBodies(float4* newPos, float4* newVel, float4* oldPos, float4* oldVel,
               float deltaTime, float damping, int numBodies)
{
    int index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    float4 pos = oldPos[index];

    float3 accel = computeBodyAccel(pos, oldPos, numBodies);
    float4 vel = oldVel[index];

    vel.x += accel.x * deltaTime;
    vel.y += accel.y * deltaTime;
    vel.z += accel.z * deltaTime;

    vel.x *= damping;
    vel.y *= damping;
    vel.z *= damping;

    // new position = old position + velocity * deltaTime
    pos.x += vel.x * deltaTime;
    pos.y += vel.y * deltaTime;
    pos.z += vel.z * deltaTime;

    // store new position and velocity
    newPos[index] = pos;
    newVel[index] = vel;
}
```

**Note that the code in the white-paper and the implementation are not quite the same!**

# Other N-body-style problems

# Particle physics

- Turbulence:  
<http://www.youtube.com/v/RuZQpWo9Qhs>
- Lava:  
<http://www.youtube.com/v/7bxP2cXVQt8>
- PhysX  
<http://www.youtube.com/v/z5Cdc2gFngk>

# CUDA PhysX

- dev'd by ETH Zurich spin-off NovodeX, acquired by Ageia, acquired by Nvidia 2008
- realtime physics engine middleware
- 150+ PC/Xbox/PS3 titles
- rigid body physics, articulated dynamics, volumetric fluid, soft bodies, cloth, etc.

# Flocking

- Individuals governed by simple rules
- Aggregated in large flocks
- Complex behaviour arises  
<http://www.youtube.com/v/XH-groCeKbE>
- Impossible to predict/calculate, but can be simulated

# “Boids”

- Classic AI paper (1987):
  - separation
  - alignment
  - cohesion
  - predator and prey
  - in-flock positioning
- <http://www.lalena.com/AI/Flock/>

# Boids on CUDA

- Dogfight

<http://www.youtube.com/watch?v=Z-gpwCspxi8>

- Large flocks

<http://www.youtube.com/watch?v=g60UrbWxt-8>

# Next lab/assignment!