Application:
all-pairs n-body problem and gravitation

Theoretical presentation

Sequential implementation

CUDA without shared memory

CUDA with shared memory

References:

http://farside.ph.utexas.edu/teaching/329/lectures/node35.html *for Runge-Kutta method*

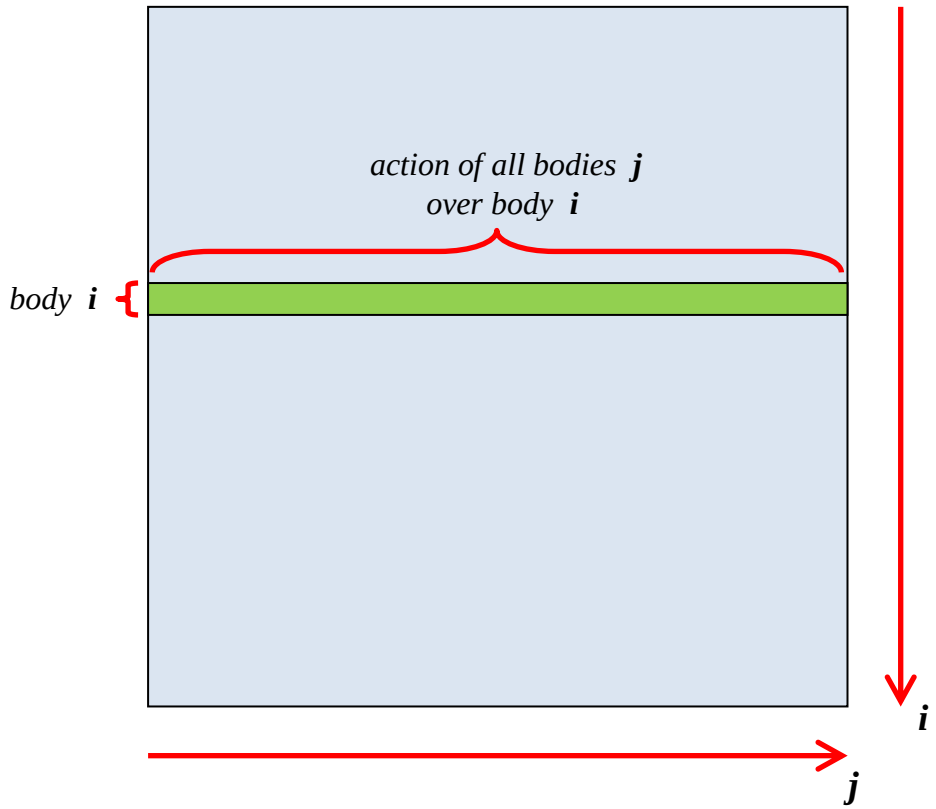
http://www.ast.cam.ac.uk/~sverre/web/pages/nbody.htm

http://www.scholarpedia.org/article/N-body_simulations_(gravitational) } *for n-body problem*

Theoretical presentation

all-pairs n-body problem:

- ***n*** *bodies fully interacting with each other:*



- *position* $\mathbf{p_i}$ *and velocity* $\mathbf{v_i}$ *of body* ***i*** *:* $pv_i = \begin{bmatrix} p_i \\ v_i \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \\ vx_i \\ vy_i \\ vz_i \end{bmatrix}$

- *time derivative of the system's global state:* $\frac{d}{dt}\begin{bmatrix} pv_0 \\ . \\ . \\ . \\ pv_{n-1} \end{bmatrix} = D\left(\begin{bmatrix} pv_0 \\ . \\ . \\ . \\ pv_{n-1} \end{bmatrix}, t\right)$

time integration:

global state **s(t)** is known at time **t**

$$\frac{d}{dt}\,s(t) = D(s(t), t)$$

→ obtain global state **s(t + Δt)** at time **t + Δt**

- *Euler method*:

$$s(t+\Delta t) = s(t) + D(s(t), t) \times \Delta t$$

inaccurate + numerical instability: errors pile up over time

- *second-order Runge-Kutta method*:

$$\begin{cases} D_1 = D(s(t), t) \\ D_2 = D(s(t) + D_1 \times \Delta t/2,\ t + \Delta t/2) \\ s(t+\Delta t) = s(t) + D_2 \times \Delta t \end{cases}$$

*first estimate of **D** (at **t**)*

*second estimate of **D** (at **t + Δt / 2**)*

accurate + numerically stable = excellent

fourth-order Runge-Kutta method is often preferred...

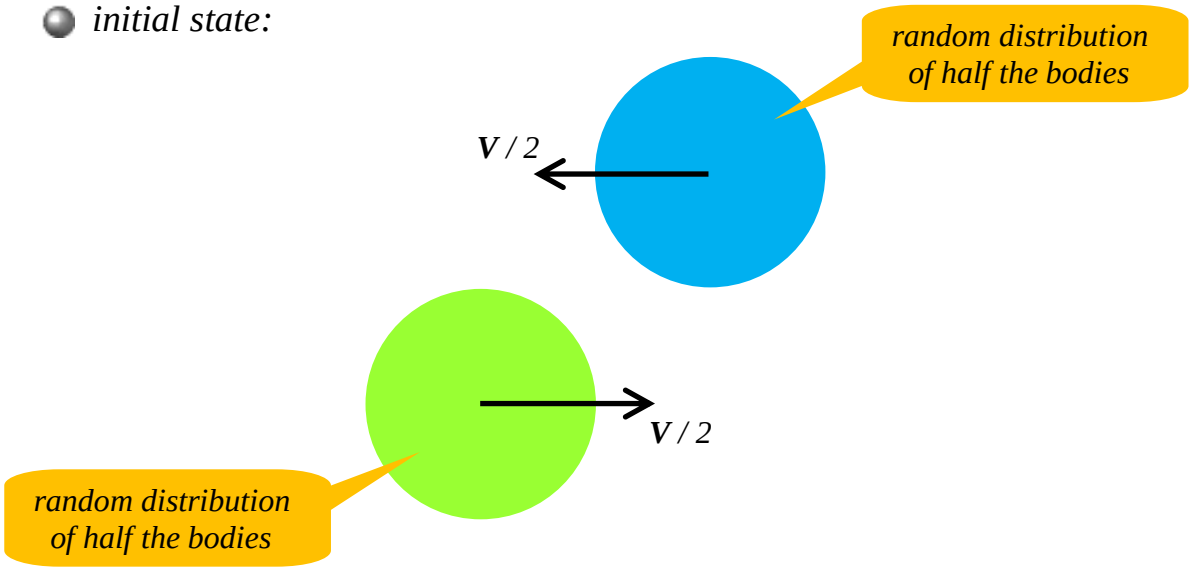
gravitational interactions:

- *body* ***i*** *of mass* m_i
- *gravitational constant* $G = 6.67384e\text{-}11$
- *softening length* ε
- *Newton's law of gravity:*

$$D(s(t), t) = \frac{d}{dt}\begin{bmatrix} \cdot \\ \cdot \\ p_i \\ v_i \\ \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \\ v_i \\ G \sum_{j \neq i} \frac{m_j (p_j - p_i)}{\left(|p_j - p_i|^2 + \varepsilon^2 \right)^{3/2}} \\ \cdot \\ \cdot \end{bmatrix}$$

case study: collision of two galaxies - simulation parameters and initial state:

- *total mass of the two galaxies = 4×10^{42} kg*
- *initial diameter of each galaxy* $\boldsymbol{H} = 10^{21}$ *m*
- *softening length* $\boldsymbol{\varepsilon} = \boldsymbol{H} / 100$
- *initial velocity between the two galaxies* $\boldsymbol{V} = 350\times10^{3}$ *m/s*
- *initial state:*



- *time step* $\boldsymbol{DT} = 10^{5}\times31536000$ *s (100,000 years)*

Sequential implementation

data structures:

```
struct position_velocity
{
   double x , y , z;
   double vx , vy , vz;
};


struct state
{
   int n;
   double t;

   double *m;

   struct position_velocity *pv;

   struct position_velocity *dpv , *pv_;
};
```

main function:

*state = external variable **S***
→ can be accessed within
display** and **evolve

```
struct state *S;

int main(int argc, char *argv[])
{
   int n = atoi(argv[1]);

   S = malloc_state(n);
   init_state(S , n);

   glutInit(&argc, argv);    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(1000, 1000);    glutInitWindowPosition(0, 0);
   glutCreateWindow ("nbody");
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glutReshapeFunc(reshape);    glutDisplayFunc(display);    glutIdleFunc(evolve);
   glutMainLoop();

   return 0;
}
```

GLUT callback functions:

```
void reshape(int w, int h)
{
   glViewport (0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(  - HVIEW / 2, HVIEW / 2
           , - HVIEW / 2 , HVIEW / 2
           , 0.0 , HVIEW);
}



void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glBegin(GL_POINTS);
      for (int i = 0 ; i < S->n ; i++)
      {
         if (i < S->n / 2)  glColor3f(1.0 , 1.0 , 0.0);
                      else  glColor3f(1.0 , 0.5 , 1.0);

         struct position_velocity *pvi = S->pv + i;
         glVertex3f(pvi->x , pvi->y , pvi->z);
      }
   glEnd();

   glutSwapBuffers();
}



void evolve(void)
{
   evolve_state_over_time_step(S , DT);

   glutPostRedisplay();
}
```

time integration:

```
void evolve_state_over_time_step(struct state *s , double dt)
{
   derivatives_positions_velocities(s->pv , s);

   increment_positions_velocities(s->pv_ , s->pv , s->dpv , dt / 2 , s->n);

   derivatives_positions_velocities(s->pv_ , s);

   increment_positions_velocities(s->pv , s->pv , s->dpv , dt , s->n);

   s->t += dt;
}
```

D_1

$s(t) + D_1 \times \Delta t / 2$

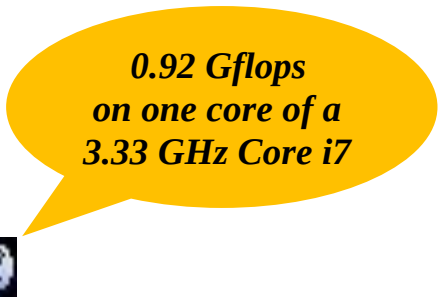
D_2

$s(t) + D_2 \times \Delta t$

```
void increment_positions_velocities(   struct position_velocity *pv1
                                     , struct position_velocity *pv0
                                     , struct position_velocity *dpv
                                     , double k , int n)
{
   for (int i = 0 ; i < n ; i++)
   {
      struct position_velocity *pv1i = pv1 + i , *pv0i = pv0 + i , *dpvi = dpv + i;

      pv1i->x = pv0i->x + dpvi->x * k;
      pv1i->y = pv0i->y + dpvi->y * k;
      pv1i->z = pv0i->z + dpvi->z * k;

      pv1i->vx = pv0i->vx + dpvi->vx * k;
      pv1i->vy = pv0i->vy + dpvi->vy * k;
      pv1i->vz = pv0i->vz + dpvi->vz * k;
   }
}
```

time derivatives of positions and velocities:

```
//---- calculate s->dpv for pv = s->pv or s->pv_
void derivatives_positions_velocities(struct position_velocity *pv , struct state *s)
{
   double pipjx , pipjy , pipjz , d2 , q;

   for (int i = 0 ; i < s->n ; i++)
   {
      struct position_velocity *pvi = pv + i , *dpvi = s->dpv + i;

      dpvi->x = pvi->vx;    dpvi->vx = 0.0;
      dpvi->y = pvi->vy;    dpvi->vy = 0.0;
      dpvi->z = pvi->vz;    dpvi->vz = 0.0;

      for (int j = 0 ; j < s->n ; j++)
         if (j != i)
         {
            struct position_velocity *pvj = pv + j ;

            pipjx = pvj->x - pvi->x;
            pipjy = pvj->y - pvi->y;
            pipjz = pvj->z - pvi->z;
            d2 = pipjx * pipjx + pipjy * pipjy + pipjz * pipjz + EPS * EPS;
            q = s->m[j] / d2 / sqrt(d2);

            dpvi->vx += q * pipjx;
            dpvi->vy += q * pipjy;
            dpvi->vz += q * pipjz;
         }

      dpvi->vx *= G;
      dpvi->vy *= G;
      dpvi->vz *= G;
   }
}
```

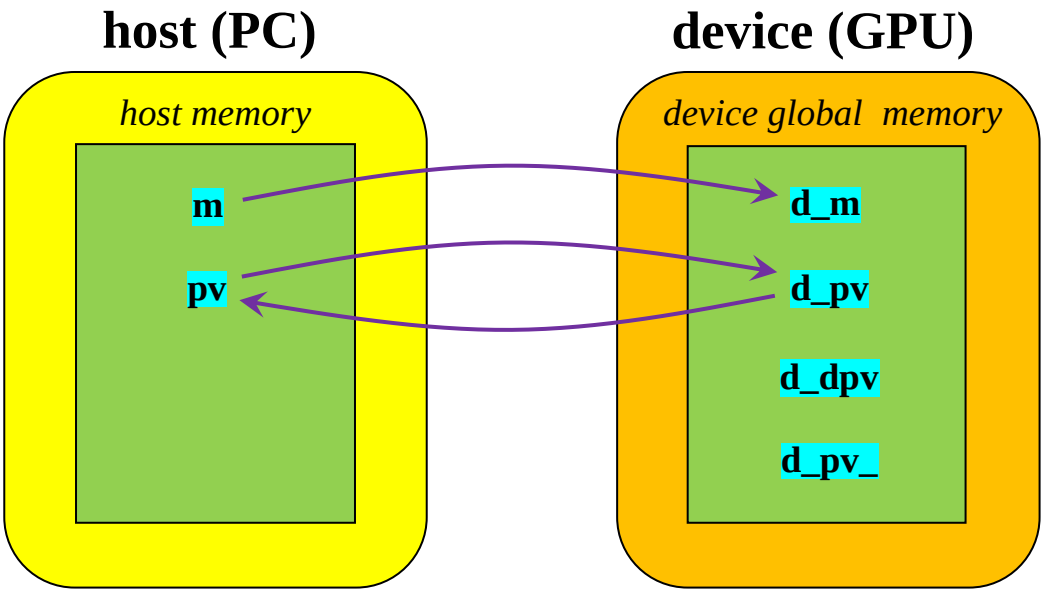


sqrt is counted as 4 floating-point operations

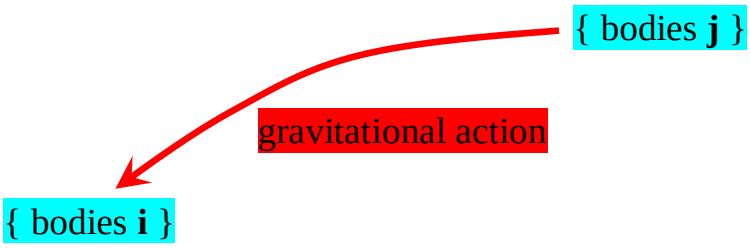
CUDA without shared memory

data structures:

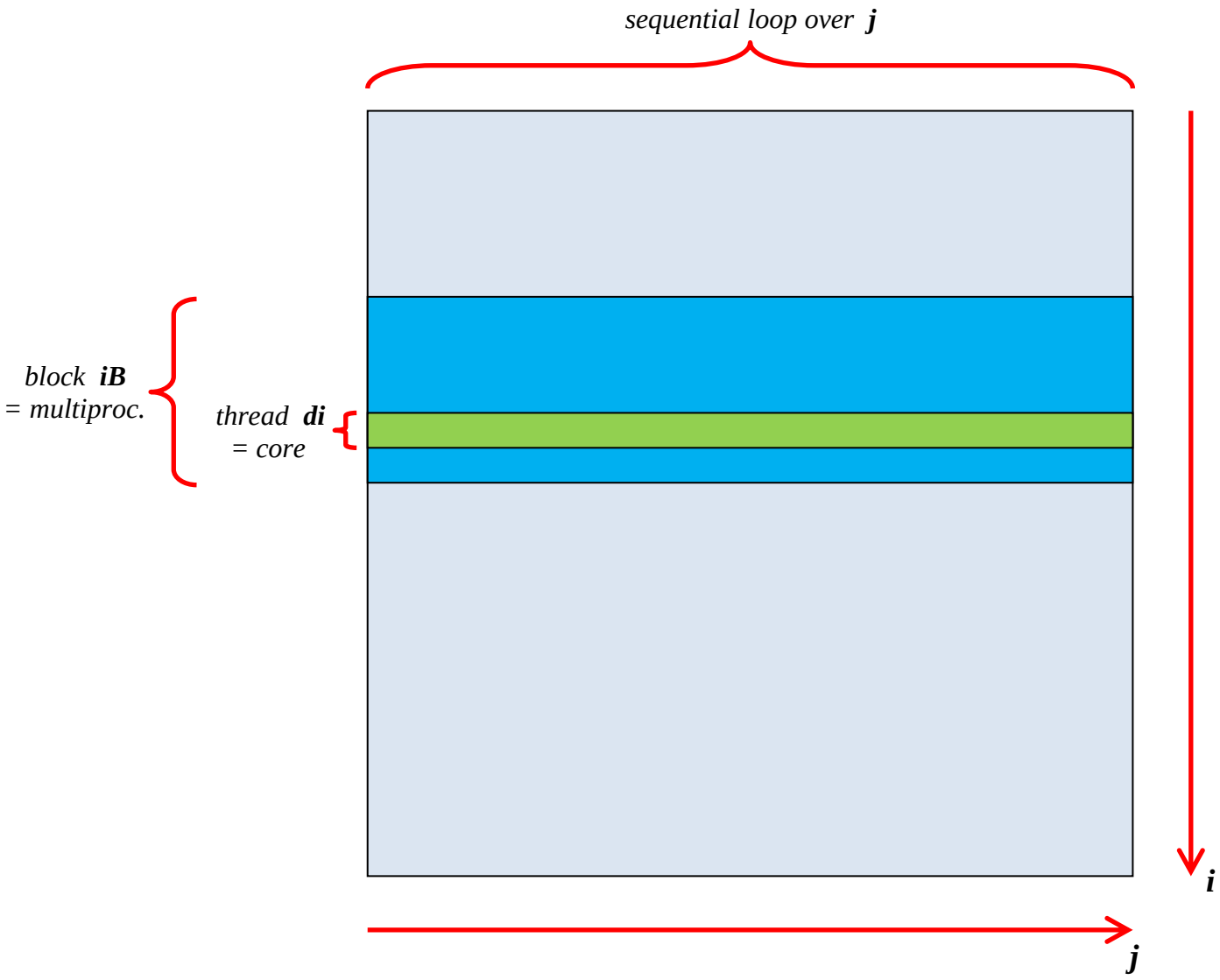
```
struct state
{
   int n;
   double t;

   double *m , *d_m;

   struct position_velocity *pv , *d_pv;

   struct position_velocity *d_dpv , *d_pv_;
};
```



data partitioning:



→ 1D partitioning of { bodies **i** }



CUDA parallelization of **increment_positions_velocities :**

```
void increment_positions_velocities(   struct position_velocity *d_pv1
                                     , struct position_velocity *d_pv0
                                     , struct position_velocity *d_dpv
                                     , double k , int n)
{
   int nB   =  n  / SB + (n  % SB ? 1 : 0);

   kernel_inc_pv <<< nB , SB >>> (d_pv1 , d_pv0 , d_dpv , k , n , nB);

   CUDA_CALL("wait kernel_inc_pv" , cudaThreadSynchronize());
}
```

```
__global__ void kernel_inc_pv(   struct position_velocity *d_pv1
                               , struct position_velocity *d_pv0
                               , struct position_velocity *d_dpv
                               , double k , int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;     int i = i0 + threadIdx.x;

   if (i < i1)
   {
      struct position_velocity *pv1i = d_pv1 + i , *pv0i = d_pv0 + i , *dpvi = d_dpv + i;

      pv1i->x = pv0i->x + dpvi->x * k;
      pv1i->y = pv0i->y + dpvi->y * k;
      pv1i->z = pv0i->z + dpvi->z * k;

      pv1i->vx = pv0i->vx + dpvi->vx * k;
      pv1i->vy = pv0i->vy + dpvi->vy * k;
      pv1i->vz = pv0i->vz + dpvi->vz * k;
   }
}
```

CUDA parallelization of **derivatives_positions_velocities :**

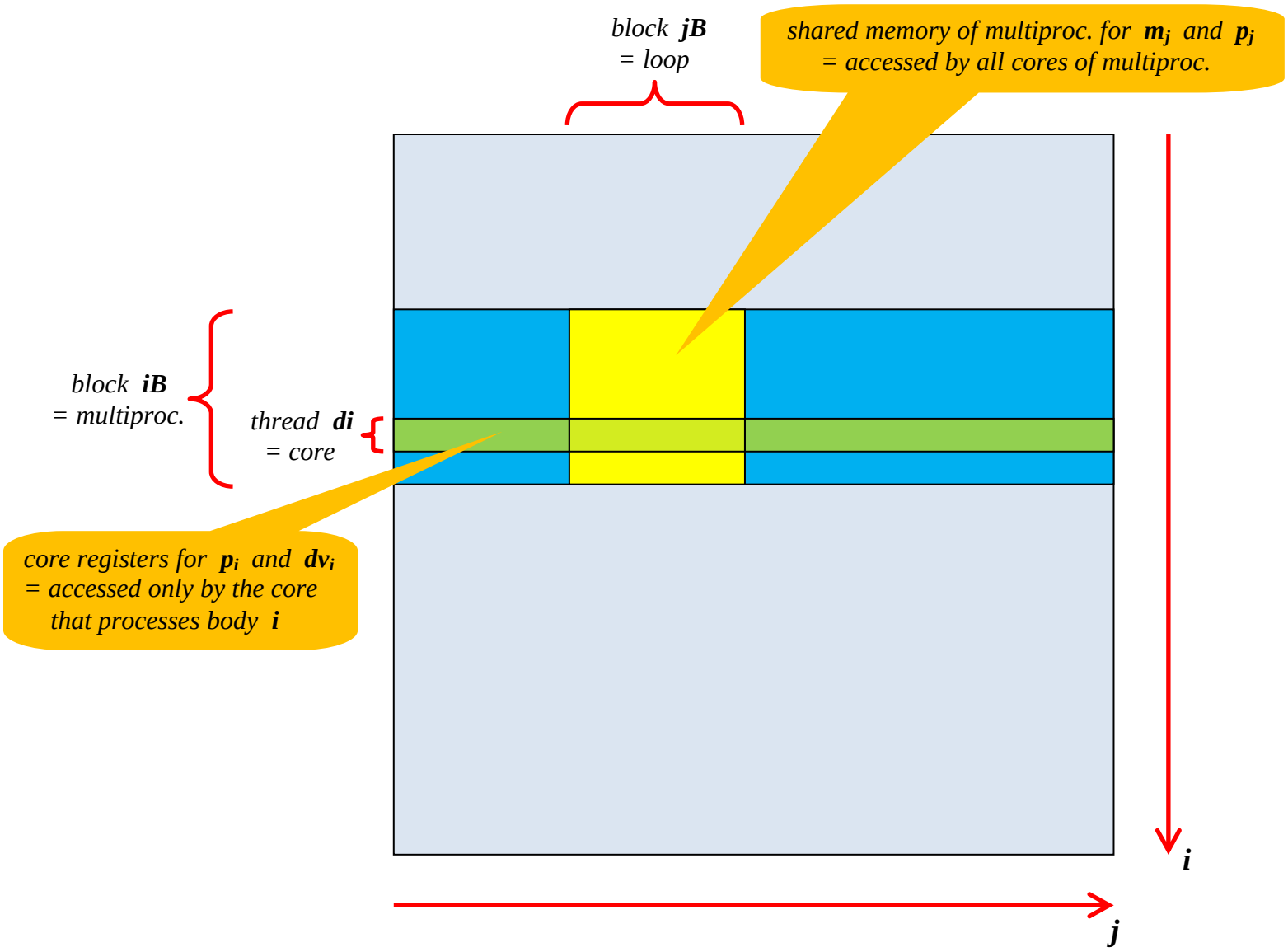
```
//---- calculate s->d_dpv for d_pv = s->d_pv or s->d_pv_
void derivatives_positions_velocities(struct position_velocity *d_pv , struct state *s)
{
   int nB   =  s->n  / SB + (s->n  % SB ? 1 : 0);

   kernel_deriv <<< nB , SB >>> (s->d_m , d_pv , s->d_dpv , EPS * EPS , G , s->n , nB);

   CUDA_CALL("wait kernel_deriv" , cudaThreadSynchronize());
}
```

```
__global__ void kernel_deriv(   double *d_m
                              , struct position_velocity *d_pv
                              , struct position_velocity *d_dpv
                              , double eps2 , double G , int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n / nB , i1 = (iB + 1) * n / nB;    int i = i0 + threadIdx.x;
   double pipjx , pipjy , pipjz , d2 , q;

   if (i < i1)
   {
      struct position_velocity *pvi = d_pv + i , *dpvi = d_dpv + i;

      dpvi->x = pvi->vx;    dpvi->vx = 0.0;
      dpvi->y = pvi->vy;    dpvi->vy = 0.0;
      dpvi->z = pvi->vz;    dpvi->vz = 0.0;

      for (int j = 0 ; j < n ; j++)
         if (j != i)
         {
            struct position_velocity *pvj = d_pv + j ;

            pipjx = pvj->x - pvi->x;
            pipjy = pvj->y - pvi->y;
            pipjz = pvj->z - pvi->z;
            d2 = pipjx * pipjx + pipjy * pipjy + pipjz * pipjz + eps2;
            q = d_m[j] / d2 / sqrt(d2);
            dpvi->vx += q * pipjx;
            dpvi->vy += q * pipjy;
            dpvi->vz += q * pipjz;
         }

      dpvi->vx *= G;
      dpvi->vy *= G;
      dpvi->vz *= G;
   }
}
```

up to 26 Gflops

SB = 400 →

CUDA with shared memory



for each thread of each multiprocessor:

- current thread loads p_i into core registers
- for each block jB :
  - all threads of multiprocessor load m_j and p_j into shared memory
  - synchronize all threads of multiprocessor
  - current thread calculates action of body **j** over body **i**
  - synchronize all threads of multiprocessor
- current thread retrieves dv_i from core registers

```
__global__ void kernel_deriv(   double *d_m
                              , struct position_velocity *d_pv
                              , struct position_velocity *d_dpv
                              , double eps2 , double G , int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n / nB , i1 = (iB + 1) * n / nB;    int di = threadIdx.x , i = i0 + di;
   int jB , j0 , j1 , dj , j;
   double pipjx , pipjy , pipjz , d2 , q;
   double  xi , yi , zi , dvxi , dvyi , dvzi;
   __shared__  double  Mj[SB] , Xj[SB] , Yj[SB] , Zj[SB];

   if (i < i1)                //----  for body i, thread  di  loads  xi , yi , zi  and sets  dvxi , dvyi , dvzi  at  0
   {
      struct position_velocity *pvi = d_pv + i , *dpvi = d_dpv + i;
      dpvi->x = pvi->vx;   xi = pvi->x;    dvxi = 0;
      dpvi->y = pvi->vy;   yi = pvi->y;    dvyi = 0;
      dpvi->z = pvi->vz;   zi = pvi->z;    dvzi = 0;
   }

   for (jB = 0 ; jB < nB ; jB++)
   {
      j0 = jB * n / nB;    j1 = (jB + 1) * n / nB;

      j = j0 + di;                //----  for body  j = j0 + di, thread  di  loads  Mj , Xj , Yj , Zj
      if (j < j1)
      {
         Mj[di] = d_m[j];
         struct position_velocity *pvj = d_pv + j;
         Xj[di] = pvj->x;    Yj[di] = pvj->y;    Zj[di] = pvj->z;
      }
      __syncthreads();

      if (i < i1)
         for (j = j0 ; j < j1 ; j++)
         {  dj = j - j0;

            if (j != i)
            {
               pipjx = Xj[dj] - xi;    pipjy = Yj[dj] - yi;    pipjz = Zj[dj] - zi;
               d2 = pipjx * pipjx + pipjy * pipjy + pipjz * pipjz + eps2;
               q = Mj[dj] / d2 / sqrt(d2);
               dvxi += q * pipjx;    dvyi += q * pipjy;    dvzi += q * pipjz;
            }
         }
      __syncthreads();
   }

   if (i < i1)                //----  for body  i, thread  di  stores  dvxi , dvyi , dvzi
   {
      struct position_velocity *dpvi = d_dpv + i;
      dpvi->vx = G * dvxi;    dpvi->vy = G * dvyi;    dpvi->vz = G * dvzi;
   }
}
```

up to 45 Gflops
shared memory → × 1.7

example: simulation with n = 32000