# Семинар 9: Задача N тел

## 1 Взаимодействие тел

```
__device__ float3
bodyBodyInteraction(float3 ai, float4 bi, float4 bj)
{
    float3 r;

    // r_ij  [3 FLOPS]
    r.x = bi.x - bj.x;
    r.y = bi.y - bj.y;
    r.z = bi.z - bj.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;
    distSqr += softeningSquared;

    // invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
    float invDist = rsqrtf(distSqr);
float invDistCube =  invDist * invDist * invDist;

    //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;

    return ai;
}
```

## 2 Гравитационный потенциал

```
// This is the "tile_calculation" function from the GPUG3 article.
__device__ float3 gravitation(float4 myPos, float3 accel)
{
    extern __shared__ float4 sharedPos[];

    // The CUDA 1.1 compiler cannot determine that i is not going to
    // overflow in the loop below.  Therefore if int is used on 64-bit linux
    // or windows (or long instead of long long on win64), the compiler
    // generates suboptimal code.  Therefore we use long long on win64 and
    // long on everything else. (Workaround for Bug ID 347697)
#ifdef _Win64
    unsigned long long i = 0;
#else
```

```
    unsigned long i = 0;
#endif

    // Here we unroll the loop to reduce bookkeeping instruction overhead
    // 32x unrolling seems to provide best performance

    // Note that having an unsigned int loop counter and an unsigned
    // long index helps the compiler generate efficient code on 64-bit
    // OSes.  The compiler can't assume the 64-bit index won't overflow
    // so it incurs extra integer operations.  This is a standard issue
    // in porting 32-bit code to 64-bit OSes.

#pragma unroll 32
    for (unsigned int counter = 0; counter < blockDim.x; counter++ )
    {
        accel = bodyBodyInteraction(accel, SX(i++), myPos);
    }

    return accel;
}
```

# 3 Вычисление ускорения

```
template <bool multithreadBodies>
__device__ float3
computeBodyAccel(float4 bodyPos, float4* positions, int numBodies)
{
    extern __shared__ float4 sharedPos[];
```

```
    float3 acc = {0.0f, 0.0f, 0.0f};

    int p = blockDim.x;
    int q = blockDim.y;
    int n = numBodies;
    int numTiles = n / (p * q);

    for (int tile = blockIdx.y; tile < numTiles + blockIdx.y; tile++)
    {
        sharedPos[threadIdx.x+blockDim.x*threadIdx.y] =
            multithreadBodies ?
            positions[WRAP(blockIdx.x + q * tile +
threadIdx.y, gridDim.x) * p + threadIdx.x] :
            positions[WRAP(blockIdx.x + tile,
gridDim.x) * p + threadIdx.x];

        __syncthreads();

        // This is the "tile_calculation" function from the GPUG3 article.
        acc = gravitation(bodyPos, acc);

        __syncthreads();
    }

    if (multithreadBodies)
    {
        SX_SUM(threadIdx.x, threadIdx.y).x = acc.x;
```

```
        SX_SUM(threadIdx.x, threadIdx.y).y = acc.y;
        SX_SUM(threadIdx.x, threadIdx.y).z = acc.z;

        __syncthreads();

        // Save the result in global memory for the integration step
        if (threadIdx.y == 0)
        {
            for (int i = 1; i < blockDim.y; i++)
            {
                acc.x += SX_SUM(threadIdx.x,i).x;
                acc.y += SX_SUM(threadIdx.x,i).y;
                acc.z += SX_SUM(threadIdx.x,i).z;
            }
        }
    }

    return acc;
```

# 4 Интегрирование уравнений движения

```
template<bool multithreadBodies>
__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<multithreadBodies>(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;
}
```