```
/*
 *  This is a commented version of the original N-Body
 *  program, some of the internal variable has a different
 *  but better readable name. The structure and principle
 *  of this program has no big change.
 *
 *  Commented By Daniel Zhang <danielzhang0212@gmail.com>
 */


#include <stdio.h>
#include <cmath>
#include <cassert>
#include <cutil.h>
#include <omp.h>
#include "cuda_pointer.h"


/*
 * define some constants, like
 * the number of thread of a block,
 * j blocks, and i blocks
 */
#define NTHREAD 64 // 64, 96, 128 or 192
#define NJBLOCK 16 // 8800GTS/512 has 16
#define NIBLOCK 16 // 16 or 32
#define NIMAX (NTHREAD * NIBLOCK) // 1024
#define NBMAX 128 // NNB per block

/*
 *  user define vector
 */
template <class T>
struct myvector{
        int num;
        T *val;
        myvector(){
                num = 0;
                val = NULL;
        }
        ~myvector(){
                delete [] val;
        }
        void clear(){
                num = 0;
        }
        void reserve(size_t count){
                val = new T[count];
        }
        void free(){
                delete [] val;
        }
        void push_back(const T &t){
                val[num++] = t;
        }
        size_t size(){
                return num;
        }
        T &operator[](int i){
                return val[i];
        }
};


/*
 * define the time related functions
 * in order to record the calculation time.
 */
#include <sys/time.h>
```



```
static double get_wtime(){
        struct timeval tv;
        gettimeofday(&tv, NULL);
        return tv.tv_sec + 1.e-6 * tv.tv_usec;
}

static double time_send, time_grav;
static long long numInter


/*
 * define the structure of particle J
 * contain: position, velocity, mass and pad
 */
struct Jparticle{
        float3 pos;             // position
        float3 vel;             // velocity
        float  mass;    // mass
        float  pad;             // TODO:don't know!
        Jparticle() {}
        Jparticle(double mj, double posj[3], double velj[3]){
                pos.x = posj[0];
                pos.y = posj[1];
                pos.z = posj[2];
                mass  = mj;
                vel.x = velj[0];
                vel.y = velj[1];
                vel.z = velj[2];
        }
};


/*
 *  define the structure of particle I
 *  contain: position, velocity, pad, and h2
 */
struct Iparticle{
        float3 pos;             // position
        float3 vel;             // velocity
        float  pad;             // TODO:don't know
        float  h2;              // Threshold to dicide a neighbor
        Iparticle() {}
        Iparticle(double h2i, double posi[3], double veli[3]){
                pos.x = posi[0];
                pos.y = posi[1];
                pos.z = posi[2];
                h2    = h2i;
                vel.x = veli[0];
                vel.y = veli[1];
                vel.z = veli[2];
        }
};

/*
 * define the structure of the force
 * between two particles, including jerk
 * force, acceleration, potential, and neighborhood
 */
struct Force{
        float3 acc;             // acceleration
        float3 jrk;             // jerk force
        float  pot;             // potential
        int    nnb;     // number of neighbor particles ONE BYTE
        unsigned short  neib[NBMAX]; // 24 words
        __device__  Force(){    // created inside a kernel
                acc.x = acc.y = acc.z = 0.f;
                jrk.x = jrk.y = jrk.z = 0.f;
                pot = 0.f;
                nnb = 0;
```

```
        }
};


/*
 * calculate the FORCE contents
 * between two particles.
 * using the position and velocity of
 * two particles.
 */
__device__ void force_kernel(
                                const int id,
                                const Iparticle &ip,
                                const Jparticle &jp,
                                Force &force)
{
        float dx = jp.pos.x - ip.pos.x;         // position diff of x
        float dy = jp.pos.y - ip.pos.y;         // position diff of y
        float dz = jp.pos.z - ip.pos.z;         // position diff of z
        float dvx = jp.vel.x - ip.vel.x;        // velocity diff of x
        float dvy = jp.vel.y - ip.vel.y;        // velocity diff of y
        float dvz = jp.vel.z - ip.vel.z;        // velocity diff of z

        float dist_square = dx*dx + dy*dy + dz*dz;

        float inverse = rsqurtf(dist_square);   // used as denominator

        if(dist_square < ip.h2) // if the distance is smaller than a value
        {
                // add this particle into neibour list
                force.neib[force.nnb & (NBMAX-1)] = (unsigned)j;        // do mo
dulus
                /* NBMAX-1=(11111111) in binary */
                force.nnb++;

                /* neighbor's force does not count */
                inverse = 0.f;
        }

        float inverse_square = inverse * inverse;
        float potential = jp.mass * inverse;            // potential
        float acc_para = mass_inverse * inverse;        // accelerate para

        rv = -3.f * inverse_square * (dx*dvx + dy*dvy + dz*dvz);

        /* the final calculation of the FORCE */
#ifdef POTENTIAL
        force.pot += potential;
#endif
        force.acc.x += acc_para * dx;
        force.acc.y += acc_para * dy;
        force.acc.z += acc_para * dz;
        force.jrk.x += acc_para * (dvx + rv * dx);
        force.jrk.y += acc_para * (dvy + rv * dy);
        force.jrk.z += acc_para * (dvz + rv * dz);

        // FUNCTION DONE
}

/*
 *  Main kernel of the n-body forces calculation
 */
__global__ void KERNEL(
                                int nbody,      // number of bodies
                                Iparticle ips[],
                                Jparticle jps[],
                                Force     forces[][NJBLOCK])
{
        int i_bid = blockIdx.x;
```



```
        int j_bid = blockIdx.y;
        int tid = threadIdx.x;
        int i_addr = tid + NTHREAD * i_bid;
        int j_start=(nbody/NJBLOCK) * j_bid;
        int j_end=(nbody/NJBLOCK) * j_bid +1 ;


        Iparticle ip = ips[i_addr];             // fetch the i particle
        Force fo;

        /*
         * loop every block of j particle,
         * where variable j stand for the first
         * element of the Jblock
         */
        for(int j=j_start;j<j_end;j+=NTHREAD)
        {
                __shared__ Jparticle jpshare[NTHREAD];  // use shared memory
                __syncthreads();

                /* copy data into shared memory */
                float4 *src = (float4 *) &jps[j];
                float4 *dst = (float4 *) jpshare;
                dst[       tid] = src[       tid];
                dst[NTHREAD+tid] = src[NTHREAD+tid];
                /*  TODO why copy 2 blocks?
                 */
                __syncthreads();

                /*
                 *  calculate the I particle's force,
                 *  interactive with every J particle
                 *  in the whole system.
                 */
                if(j_end-j<NTHREAD)
                {
                        for(int partId=0;partId<jend-j;partId++)
                        {
                                Jparticle jp = jpshare[partId];
                                force_kernel(partId, ip, jp, fo);
                        }
                }
                else
                {
#pragma unroll
                        for(int partId=0;partId<NTHREAD;partId++)
                        {
                                Jparticle jp = jpshare[partId];
                                force_kernel(partId, ip, jp, fo);
                        }
                }
        }
        /*
         *  the fo's calculation is complete,
         *  save it into buffer. fo stand for
         *  the i's Iparticle interactive with the
         *  j_bid's block of Jparticle
         */
        forces[i_addr][j_bid] = fo;


}

/*
 *  The functions below are CPU functions
 */

static cudaPointer <Jparticle> jps;
static cudaPointer <Iparticle> ips;
```

```
static cudaPointer <Force[NJBLOCK]> forces;

#define MAX_CPU 1
static myvector<int> nblist[MAX_CPU];
static int nbody, nbodymax;

/*
 *  the initial and end functions
 */
void GPUNB_open(int nbmax){
        time_send = time_grav = 0.0;
        numInter = 0;

        jpbuf.allocate(nbmax + NTHREAD);
        ipbuf.allocate(NIMAX);
        fobuf.allocate(NIMAX);
        nbodymax = nbmax;
#pragma omp parallel
        {
                int tid = 0;
                nblist[tid].reserve(nbmax);
        }
}
void GPUNB_close(){
        jpbuf.free();
        ipbuf.free();
        fobuf.free();
        nbodymax = 0;

#ifdef PROFILE
        fprintf(stderr, "**********************\n");
        fprintf(stderr, "time send : %f sec\n", time_send);
        fprintf(stderr, "time grav : %f sec\n", time_grav);
        fprintf(stderr, "%f Gflops (gravity part only)\n", 60.e-9 * numInter / t
ime_grav);
        fprintf(stderr, "**********************\n");
#endif
}
void GPUNB_send(
                int nj,
                double mj[],
                double xj[][3],
                double vj[][3]){
        time_send -= get_wtime();
        nbody = nj;
        assert(nbody <= nbodymax);
        for(int j=0; j<nj; j++){
                jpbuf[j] = Jparticle(mj[j], xj[j], vj[j]);
        }

        jpbuf.htod(nj);
        time_send += get_wtime();
}



/*
 *  MAIN FUNCTION of this program.
 *  Call the kernel.
 */
void GPUNB_regf(
                                int nI,                 // number of I particle
                                double h2[],    // one para of Iparticle
                                double xi[][3], // position vector
                                double vi[][3], // velocity vector
                                double acc[][3],// acceleration vecotor
                                double jrk[][3],// force vector
                                double pot[],   // potential
                                int lmax,               // TODO
```



```
                                int nbMax,              // max number of neighbo
r
                                int *listbase)  // TODO
{

        /* get initial time */
        time_grav -= get_wtime();
        numInter += ni * nbody;
        assert(0 < ni && ni <= NIMAX)

        for(int i=0;i<nI; i++)
        {
                ips[i] = Iparticle(h2[i],xi[i],vi[i]);
        }

        /* load the particles into device memory */
        ipbuf.htod(ni);

        /* start the kernel */
        int niblock = 1 + (nI - 1) / NTHREAD;
        dim3 grid(niblock , NJBLOCK, 1);
        dim3 threads(NTHREAD, 1,1,);
        KERNEL <<< grid, threads >>> (nbody,ips,jps,forces)

        forces.dtoh(nI);
        // reduction phase

#pragma omp parallel for
        for(int i=0;i<nI;i++)
        {
                // TODO I think "tid" should be "i"
                //      For it has never changed in
                //      every loop.
                int tid=0;
                double ax=0,ay=0,az=0;  // acceleration
                double jx=0,jy=0,jz=0;  // jerk force
#ifdef POTENTIAL
                double poti=0;
#endif
                for(int jblock=0;jblock<NJBLOCK;jblock++)
                {
                        Force &fo = forces[i][jblock];
                        ax += fo.acc.x;
                        ay += fo.acc.y;
                        az += fo.acc.z;
                        jx += fo.jrk.x;
                        jy += fo.jrk.y;
                        jz += fo.jrk.z;
#ifdef POTENTIAL
                        poti += fo.pot;
#endif
                }
                /* save the sum of individual result */
                acc[i][0] = ax;
                acc[i][1] = ay;
                acc[i][2] = az;
                jrk[i][0] = jx;
                jrk[i][1] = jy;
                jrk[i][2] = jz;
#ifdef POTENTIAL
                pot[i] = poti;
#endif

                /* TODO something about neighbors */
                bool overflow = false;
                for(int jblock=0;jblock<NJBLOCK;jblock++)
                {
                        Force &fo = forces[i][jblock];
                        int jstart = (nbody / NJBLOCK) * jblock;
```

```
                        if(fo.nnb <= NBMAX)
                        {
                                for(int k=0;k<f.nnb;k++)
                                {
                                        int nb = fo.neib[k];
                                        while(nb<jstart)
                                                nb += (1<<16);
                                        nblist[tid].push_back(nb);
                                }
                        }
                        else
                        {
                                overflow = true;
                        }
                }

                int * nnbp = listbase + lmax * i;       // number of neighbor pa
rticles
                int * nblistp = nnbp +1;
                int nnb = nblist[tid].size();
                if(nnb > nbmax)
                        overflow =true;

                if(overflow)
                {
                        *nnbp = -1;
                }
                else
                {
                        *nnbp = nnb;
                        for(int k=0;k<nnb;k++)
                        {
                                nblistp[k]=nblist[tid][k];
                        }
                }



        }

}

/* TODO combine programming between C and C++ */
extern "C" {
        void gpunb_open_(int *nbmax){
                GPUNB_open(*nbmax);
        }
        void gpunb_close_(){
                GPUNB_close();
        }
        void gpunb_send_(
                        int *nj,
                        double mj[],
                        double xj[][3],
                        double vj[][3]){
                GPUNB_send(*nj, mj, xj, vj);
        }
        void gpunb_regf_(
                        int *ni,
                        double h2[],
                        double xi[][3],
                        double vi[][3],
                        double acc[][3],
                        double jrk[][3],
                        double pot[],
                        int *lmax,
                        int *nbmax,
                        int *list){ // list[][lmax]
                GPUNB_regf(*ni, h2, xi, vi, acc, jrk, pot, *lmax, *nbmax, list);
```



```
        }
}
```