```
/*--------------------------------------------------------------------*/
__global__ void gpuppgrbppush23l(float ppart[], float fxy[],
                                 float bxy[], int kpic[], int noff,
                                 int nyp, float qbm, float dt,
                                 float dtc, float ci, float *ek,
                                 int idimp, int nppmx, int nx, int ny,
                                 int mx, int my, int nxv, int nypmx,
                                 int mx1, int mxyp1, int ipbc) {
/* for 2-1/2d code, this subroutine updates particle co-ordinates and
   velocities using leap-frog scheme in time and first-order linear
   interpolation in space, for relativistic particles with magnetic field
   Using the Boris Mover.
   threaded version using guard cells, for distributed data
   data read in tiles
   particles stored segmented array
   131 flops/particle, 4 divides, 2 sqrts, 25 loads, 5 stores
   input: all, output: ppart, ek
   momentum equations used are:
   px(t+dt/2) = rot(1)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(2)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(3)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fx(x(t),y(t))*dt)
   py(t+dt/2) = rot(4)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(5)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(6)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fy(x(t),y(t))*dt)
   pz(t+dt/2) = rot(7)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(8)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(9)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fz(x(t),y(t))*dt)
   where q/m is charge/mass, and the rotation matrix is given by:
      rot[0] = (1 - (om*dt/2)**2 + 2*(omx*dt/2)**2)/(1 + (om*dt/2)**2)
      rot[1] = 2*(omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
      rot[2] = 2*(-omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[3] = 2*(-omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
      rot[4] = (1 - (om*dt/2)**2 + 2*(omy*dt/2)**2)/(1 + (om*dt/2)**2)
      rot[5] = 2*(omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[6] = 2*(omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[7] = 2*(-omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[8] = (1 - (om*dt/2)**2 + 2*(omz*dt/2)**2)/(1 + (om*dt/2)**2)
   and om**2 = omx**2 + omy**2 + omz**2
   the rotation matrix is determined by:
   omx = (q/m)*bx(x(t),y(t))*gami, omy = (q/m)*by(x(t),y(t))*gami, and
   omz = (q/m)*bz(x(t),y(t))*gami,
   where gami = 1./sqrt(1.+(px(t)*px(t)+py(t)*py(t)+pz(t)*pz(t))*ci*ci)
   position equations used are:
   x(t+dt) = x(t) + px(t+dt/2)*dtg
   y(t+dt) = y(t) + py(t+dt/2)*dtg
   where dtg = dtc/sqrt(1.+(px(t+dt/2)*px(t+dt/2)+py(t+dt/2)*py(t+dt/2)+
   pz(t+dt/2)*pz(t+dt/2))*ci*ci)
   fx(x(t),y(t)), fy(x(t),y(t)), and fz(x(t),y(t))
   bx(x(t),y(t)), by(x(t),y(t)), and bz(x(t),y(t))
   are approximated by interpolation from the nearest grid points:
   fx(x,y) = (1-dy)*((1-dx)*fx(n,m)+dx*fx(n+1,m)) + dy*((1-dx)*fx(n,m+1)
```

```
      + dx*fx(n+1,m+1))
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   similarly for fy(x,y), fz(x,y), bx(x,y), by(x,y), bz(x,y)
   ppart[m][0][n] = position x of particle n in partition in tile m
   ppart[m][1][n] = position y of particle n in partition in tile m
   ppart[m][2][n] = x momentum of particle n in partition in tile m
   ppart[m][3][n] = y momentum of particle n in partition in tile m
   ppart[m][4][n] = z momentum of particle n in partition in tile m
   fxy[k][j][0] = x component of force/charge at grid (j,kk)
   fxy[k][j][1] = y component of force/charge at grid (j,kk)
   fxy[k][j][2] = z component of force/charge at grid (j,kk)
   that is, convolution of electric field over particle shape,
   where kk = k + noff
   bxy[k][j][0] = x component of magnetic field at grid (j,kk)
   bxy[k][j][1] = y component of magnetic field at grid (j,kk)
   bxy[k][j][2] = z component of magnetic field at grid (j,kk)
   that is, the convolution of magnetic field over particle shape,
   where kk = k + noff
   kpic = number of particles per tile
   noff = lowermost global gridpoint in particle partition.
   nyp = number of primary (complete) gridpoints in particle partition
   qbm = particle charge/mass ratio
   dt = time interval between successive calculations
   dtc = time interval between successive co-ordinate calculations
   ci = reciprical of velocity of light
   kinetic energy/mass at time t is also calculated, using
   ek = gami*sum((px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt)**2 +
        (py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt)**2 +
        (pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt)**2)/(1. + gami)
   idimp = size of phase space = 5
   nppmx = maximum number of particles in tile
   nx/ny = system length in x/y direction
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of field arrays, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
   mx1 = (system length in x direction - 1)/mx + 1
   mxyp1 = mx1*myp1, where myp1=(partition length in y direction-1)/my+1
   ipbc = particle boundary condition = (0,1,2,3) =
   (none,2d periodic,2d reflecting,mixed reflecting/periodic)
local data                                                 */
   int noffp, moffp, npoff, nppp, mxv;
   int mnoff, i, j, k, ii, nn, mm, nm;
   float qtmh, ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy;
   float dx, dy, dz, ox, oy, oz, acx, acy, acz, p2, gami, qtmg, dtg;
   float omxt, omyt, omzt, omt, anorm;
   float rot1, rot2, rot3, rot4, rot5, rot6, rot7, rot8, rot9;
   float x, y;
/* The sizes of the shared memory arrays are as follows: */
/* float sfxy[3*(mx+1)*(my+1)], sbxy[3*(mx+1)*(my+1)];   */
/* float sek[blockDim.x];                                */
/* to conserve memory, sek overlaps with sfxy and sbxy   */
/* and the name sfxy is used instead of sek              */
   float *sbxy;
   extern __shared__ float sfxy[];
```

```
   sbxy = &sfxy[3*(mx+1)*(my+1)];
   double sum1;
   qtmh = 0.5f*qbm*dt;
   ci2 = ci*ci;
   sum1 = 0.0;
/* set boundary values */
   edgelx = 0.0f;
   edgely = 1.0f;
   edgerx = (float) (nx);
   edgery = (float) (ny-1);
   if ((ipbc==2) || (ipbc==3)) {
      edgelx = 1.0f;
      edgerx = (float) (nx-1);
   }
   mxv = mx + 1;
/* k = tile number */
   k = blockIdx.x + gridDim.x*blockIdx.y;
/* loop over tiles */
   if (k < mxyp1) {
      noffp = k/mx1;
      moffp = my*noffp;
      noffp = mx*(k - mx1*noffp);
      nppp = kpic[k];
      mnoff = moffp + noff;
      npoff = idimp*nppmx*k;
/* load local fields from global array */
      nn = (mx < nx-noffp ? mx : nx-noffp) + 1;
      mm = (my < nyp-moffp ? my : nyp-moffp) + 1;
      ii = threadIdx.x;
      while (ii < mxv*(my+1)) {
         j = ii/mxv;
         i = ii - mxv*j;
         if ((i < nn) && (j < mm)) {
            sfxy[3*ii] = fxy[3*(i+noffp+nxv*(j+moffp))];
            sfxy[1+3*ii] = fxy[1+3*(i+noffp+nxv*(j+moffp))];
            sfxy[2+3*ii] = fxy[2+3*(i+noffp+nxv*(j+moffp))];
         }
         ii += blockDim.x;
      }
      ii = threadIdx.x;
      while (ii < mxv*(my+1)) {
         j = ii/mxv;
         i = ii - mxv*j;
         if ((i < nn) && (j < mm)) {
            sbxy[3*ii] = bxy[3*(i+noffp+nxv*(j+moffp))];
            sbxy[1+3*ii] = bxy[1+3*(i+noffp+nxv*(j+moffp))];
            sbxy[2+3*ii] = bxy[2+3*(i+noffp+nxv*(j+moffp))];
         }
         ii += blockDim.x;
      }
/* synchronize threads */
      __syncthreads();
/* loop over particles in tile */
      j = threadIdx.x;
```

```
      while (j < nppp) {
/* find interpolation weights */
         x = ppart[j+npoff];
         y = ppart[j+nppmx+npoff];
         nn = x;
         mm = y;
         dxp = x - (float) nn;
         dyp = y - (float) mm;
         nm = 3*(nn - noffp) + 3*mxv*(mm - mnoff);
         amx = 1.0f - dxp;
         amy = 1.0f - dyp;
/* find electric field */
         nn = nm;
         dx = amx*sfxy[nn];
         dy = amx*sfxy[nn+1];
         dz = amx*sfxy[nn+2];
         mm = nn + 3;
         dx = amy*(dxp*sfxy[mm] + dx);
         dy = amy*(dxp*sfxy[mm+1] + dy);
         dz = amy*(dxp*sfxy[mm+2] + dz);
         nn += 3*mxv;
         acx = amx*sfxy[nn];
         acy = amx*sfxy[nn+1];
         acz = amx*sfxy[nn+2];
         mm = nn + 3;
         dx += dyp*(dxp*sfxy[mm] + acx);
         dy += dyp*(dxp*sfxy[mm+1] + acy);
         dz += dyp*(dxp*sfxy[mm+2] + acz);
/* find magnetic field */
         nn = nm;
         ox = amx*sbxy[nn];
         oy = amx*sbxy[nn+1];
         oz = amx*sbxy[nn+2];
         mm = nn + 3;
         ox = amy*(dxp*sbxy[mm] + ox);
         oy = amy*(dxp*sbxy[mm+1] + oy);
         oz = amy*(dxp*sbxy[mm+2] + oz);
         nn += 3*mxv;
         acx = amx*sbxy[nn];
         acy = amx*sbxy[nn+1];
         acz = amx*sbxy[nn+2];
         mm = nn + 3;
         ox += dyp*(dxp*sbxy[mm] + acx);
         oy += dyp*(dxp*sbxy[mm+1] + acy);
         oz += dyp*(dxp*sbxy[mm+2] + acz);
/* calculate half impulse */
         dx *= qtmh;
         dy *= qtmh;
         dz *= qtmh;
/* half acceleration */
         acx = ppart[j+2*nppmx+npoff] + dx;
         acy = ppart[j+3*nppmx+npoff] + dy;
         acz = ppart[j+4*nppmx+npoff] + dz;
/* find inverse gamma */
```

```
         p2 = acx*acx + acy*acy + acz*acz;
         gami = 1.0f/sqrtf(1.0f + p2*ci2);
/* renormalize magnetic field */
         qtmg = qtmh*gami;
/* time-centered kinetic energy */
         sum1 += gami*p2/(1.0f + gami);
/* calculate cyclotron frequency */
         omxt = qtmg*ox;
         omyt = qtmg*oy;
         omzt = qtmg*oz;
/* calculate rotation matrix */
         omt = omxt*omxt + omyt*omyt + omzt*omzt;
         anorm = 2.0f/(1.0f + omt);
         omt = 0.5f*(1.0f - omt);
         rot4 = omxt*omyt;
         rot7 = omxt*omzt;
         rot8 = omyt*omzt;
         rot1 = omt + omxt*omxt;
         rot5 = omt + omyt*omyt;
         rot9 = omt + omzt*omzt;
         rot2 = omzt + rot4;
         rot4 -= omzt;
         rot3 = -omyt + rot7;
         rot7 += omyt;
         rot6 = omxt + rot8;
         rot8 -= omxt;
/* new momentum */
         dx += (rot1*acx + rot2*acy + rot3*acz)*anorm;
         dy += (rot4*acx + rot5*acy + rot6*acz)*anorm;
         dz += (rot7*acx + rot8*acy + rot9*acz)*anorm;
         ppart[j+2*nppmx+npoff] = dx;
         ppart[j+3*nppmx+npoff] = dy;
         ppart[j+4*nppmx+npoff] = dz;
/* update inverse gamma */
         p2 = dx*dx + dy*dy + dz*dz;
         dtg = dtc/sqrtf(1.0f + p2*ci2);
/* new position */
         dx = x + dx*dtg;
         dy = y + dy*dtg;
/* reflecting boundary conditions */
         if (ipbc==2) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
               ppart[j+2*nppmx+npoff] = -ppart[j+2*nppmx+npoff];
            }
            if ((dy < edgely) || (dy >= edgery)) {
               dy = ppart[j+nppmx+npoff];
               ppart[j+3*nppmx+npoff] = -ppart[j+3*nppmx+npoff];
            }
         }
/* mixed reflecting/periodic boundary conditions */
         else if (ipbc==3) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
```

```
               ppart[j+2*nppmx+npoff] = -ppart[j+2*nppmx+npoff];
            }
         }
/* set new position */
         ppart[j+npoff] = dx;
         ppart[j+nppmx+npoff] = dy;
         j += blockDim.x;
      }
/* synchronize threads */
      __syncthreads();
/* add kinetic energies in tile */
      sfxy[threadIdx.x] = (float) sum1;
/* synchronize threads */
      __syncthreads();
      lsum2(sfxy,blockDim.x);
/* normalize kinetic energy of tile */
      if (threadIdx.x==0) {
         ek[k] = sfxy[0];
      }
   }
   return;
}
```

```
/*--------------------------------------------------------------------*/
__global__ void gpu2ppgppost2l(float ppart[], float q[], int kpic[],
                               int noff, float qm, int idimp, int nppmx,
                               int mx, int my, int nxv, int nypmx,
                               int mx1, int mxyp1) {
/* for 2d code, this subroutine calculates particle charge density
   using first-order linear interpolation, periodic boundaries
   threaded version using guard cells, for distributed data
   data deposited in tiles
   particles stored segmented array
   17 flops/particle, 6 loads, 4 stores
   input: all, output: q
   charge density is approximated by values at the nearest grid points
   q(n,m)=qm*(1.-dx)*(1.-dy)
   q(n+1,m)=qm*dx*(1.-dy)
   q(n,m+1)=qm*(1.-dx)*dy
   q(n+1,m+1)=qm*dx*dy
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   ppart[m][n][0] = position x of particle n in partition in tile m
   ppart[m][n][1] = position y of particle n in partition in tile m
   q[k][j] = charge density at grid point (j,kk),
   where kk = k + noff
   kpic = number of particles per tile
   noff = lowermost global gridpoint in particle partition.
   qm = charge on particle, in units of e
   idimp = size of phase space = 4
   nppmx = maximum number of particles in tile
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of charge array, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
   mx1 = (system length in x direction - 1)/mx + 1
   mxyp1 = mx1*myp1, where myp1=(partition length in y direction-1)/my+1
local data                                                            */
   int noffp, moffp, npoff, nppp, mxv;
   int mnoff, i, j, k, ii, nn, np, mm, mp;
   float dxp, dyp, amx, amy;
/* The size of the shared memory array is as follows: */
/* float sq[(mx+1)*(my+1)]                            */
   extern __shared__ float sq[];
   mxv = mx + 1;
/* k = tile number */
   k = blockIdx.x + gridDim.x*blockIdx.y;
/* loop over tiles */
   if (k < mxyp1) {
      noffp = k/mx1;
      moffp = my*noffp;
      noffp = mx*(k - mx1*noffp);
      nppp = kpic[k];
      npoff = idimp*nppmx*k;
      mnoff = moffp + noff;
/* zero out local accumulator */
      i = threadIdx.x;
      while (i < mxv*(my+1)) {
         sq[i] = 0.0f;
```

```
         i += blockDim.x;
      }
/* synchronize threads */
       __syncthreads();
/* loop over particles in tile */
      j = threadIdx.x;
      while (j < nppp) {
/* find interpolation weights */
         dxp = ppart[j+npoff];
         nn = dxp;
         dyp = ppart[j+npoff+nppmx];
         mm = dyp;
         dxp = qm*(dxp - (float) nn);
         dyp = dyp - (float) mm;
         nn = nn - noffp;
         mm = mxv*(mm - mnoff);
         amx = qm - dxp;
         mp = mm + mxv;
         amy = 1.0f - dyp;
         np = nn + 1;
/* deposit charge within tile to local accumulator */
/* original deposit charge, has data hazard on GPU */
/*       sq[np+mp] += dxp*dyp; */
/*       sq[nn+mp] += amx*dyp; */
/*       sq[np+mm] += dxp*amy; */
/*       sq[nn+mm] += amx*amy; */
/* for devices with compute capability 2.x */
         atomicAdd(&sq[np+mp],dxp*dyp);
         atomicAdd(&sq[nn+mp],amx*dyp);
         atomicAdd(&sq[np+mm],dxp*amy);
         atomicAdd(&sq[nn+mm],amx*amy);
         j += blockDim.x;
      }
/* synchronize threads */
      __syncthreads();
/* deposit charge to global array */
      nn = mxv < nxv-noffp ? mxv : nxv-noffp;
      mm = my+1 < nypmx-moffp ? my+1 : nypmx-moffp;
      ii = threadIdx.x;
      while (ii < mxv*(my+1)) {
         j = ii/mxv;
         i = ii - mxv*j;
         if ((i < nn) && (j < mm)) {
/* original deposit charge, has data hazard on GPU */
/*          q[i+noffp+nxv*(j+moffp)] += sq[ii]; */
/* for devices with compute capability 2.x */
            atomicAdd(&q[i+noffp+nxv*(j+moffp)],sq[ii]);
         }
         ii += blockDim.x;
      }
   }
   return;
}
```

```
/*--------------------------------------------------------------------*/
__global__ void gpu2pprjppost2l(float ppart[], float cu[], int kpic[],
                                int noff, float qm, float dt, float ci,
                                int nppmx, int idimp, int nx, int ny,
                                int mx, int my, int nxv, int nypmx,
                                int mx1, int mxyp1, int ipbc) {
/* for 2-1/2d code, this subroutine calculates particle current density
   using first-order linear interpolation for relativistic particles
   in addition, particle positions are advanced a half time-step
   threaded version using guard cells, for distributed data
   data deposited in tiles
   particles stored segmented array
   47 flops/particle, 1 divide, 1 sqrt, 17 loads, 14 stores
   input: all, output: ppart, cu
   current density is approximated by values at the nearest grid points
   cu(i,n,m)=qci*(1.-dx)*(1.-dy)
   cu(i,n+1,m)=qci*dx*(1.-dy)
   cu(i,n,m+1)=qci*(1.-dx)*dy
   cu(i,n+1,m+1)=qci*dx*dy
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   and qci = qm*pi*gami, where i = x,y,z
   where gami = 1./sqrt(1.+sum(pi**2)*ci*ci)
   ppart[m][0][n] = position x of particle n in partition in tile m
   ppart[m][1][n] = position y of particle n in partition in tile m
   ppart[m][2][n] = x momentum of particle n in partition in tile m
   ppart[m][3][n] = y momentum of particle n in partition in tile m
   ppart[m][4][n] = z momentum of particle n in partition in tile m
   cu[k][j][i] = ith component of current density at grid point (j,kk),
   where kk = k + noff
   kpic = number of particles per tile
   noff = lowermost global gridpoint in particle partition.
   qm = charge on particle, in units of e
   dt = time interval between successive calculations
   ci = reciprical of velocity of light
   nppmx = maximum number of particles in tile
   idimp = size of phase space = 5
   nx/ny = system length in x/y direction
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of current array, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
   mx1 = (system length in x direction - 1)/mx + 1
   mxyp1 = mx1*myp1, where myp1=(partition length in y direction-1)/my+1
   ipbc = particle boundary condition = (0,1,2,3) =
   (none,2d periodic,2d reflecting,mixed reflecting/periodic)
local data                                                            */
   int noffp, moffp, npoff, nppp, mxv;
   int mnoff, i, j, k, ii, nn, mm;
   float ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy;
   float x, y, dx, dy, vx, vy, vz, p2, gami;
/* The size of the shared memory array is as follows: */
/* float scu[3*(mx+1)*(my+1)]                         */
   extern __shared__ float scu[];
   ci2 = ci*ci;
/* set boundary values */
```

```
   edgelx = 0.0f;
   edgely = 1.0f;
   edgerx = (float) (nx);
   edgery = (float) (ny-1);
   if ((ipbc==2) || (ipbc==3)) {
      edgelx = 1.0f;
      edgerx = (float) (nx-1);
   }
   mxv = mx + 1;
/* k = tile number */
   k = blockIdx.x + gridDim.x*blockIdx.y;
/* loop over tiles */
   if (k < mxyp1) {
      noffp = k/mx1;
      moffp = my*noffp;
      noffp = mx*(k - mx1*noffp);
      nppp = kpic[k];
      mnoff = moffp + noff;
      npoff = idimp*nppmx*k;
/* zero out local accumulator */
      i = threadIdx.x;
      while (i < 3*mxv*(my+1)) {
         scu[i] = 0.0f;
         i += blockDim.x;
      }
/* synchronize threads */
       __syncthreads();
/* loop over particles in tile */
      j = threadIdx.x;
      while (j < nppp) {
/* find interpolation weights */
         x = ppart[j+npoff];
         nn = x;
         y = ppart[j+nppmx+npoff];
         mm = y;
         dxp = qm*(x - (float) nn);
         dyp = y - (float) mm;
/* find inverse gamma */
         vx = ppart[j+2*nppmx+npoff];
         vy = ppart[j+3*nppmx+npoff];
         vz = ppart[j+4*nppmx+npoff];
         p2 = vx*vx + vy*vy + vz*vz;
         gami = 1.0f/sqrtf(1.0f + p2*ci2);
/* calculate weights */
         nn = 3*(nn - noffp) + 3*mxv*(mm - mnoff);
         amx = qm - dxp;
         amy = 1.0f - dyp;
/* deposit current */
         dx = amx*amy;
         dy = dxp*amy;
         vx *= gami;
         vy *= gami;
         vz *= gami;
/* original current deposit, has data hazard on GPU */
```

```
/*       scu[nn] += vx*dx;   */
/*       scu[nn+1] += vy*dx; */
/*       scu[nn+2] += vz*dx; */
/* for devices with compute capability 2.x */
         atomicAdd(&scu[nn],vx*dx);
         atomicAdd(&scu[nn+1],vy*dx);
         atomicAdd(&scu[nn+2],vz*dx);
         dx = amx*dyp;
         mm = nn + 3;
/* original current deposit, has data hazard on GPU */
/*       scu[mm] += vx*dy;   */
/*       scu[mm+1] += vy*dy; */
/*       scu[mm+2] += vz*dy; */
/* for devices with compute capability 2.x */
         atomicAdd(&scu[mm],vx*dy);
         atomicAdd(&scu[mm+1],vy*dy);
         atomicAdd(&scu[mm+2],vz*dy);
         dy = dxp*dyp;
         nn += 3*mxv;
/* original current deposit, has data hazard on GPU */
/*       scu[nn] += vx*dx;   */
/*       scu[nn+1] += vy*dx; */
/*       scu[nn+2] += vz*dx; */
/* for devices with compute capability 2.x */
         atomicAdd(&scu[nn],vx*dx);
         atomicAdd(&scu[nn+1],vy*dx);
         atomicAdd(&scu[nn+2],vz*dx);
         mm = nn + 3;
/* original current deposit, has data hazard on GPU */
/*       scu[mm] += vx*dy;   */
/*       scu[mm+1] += vy*dy; */
/*       scu[mm+2] += vz*dy; */
/* for devices with compute capability 2.x */
         atomicAdd(&scu[mm],vx*dy);
         atomicAdd(&scu[mm+1],vy*dy);
         atomicAdd(&scu[mm+2],vz*dy);
/* advance position half a time-step */
         dx = x + vx*dt;
         dy = y + vy*dt;
/* reflecting boundary conditions */
         if (ipbc==2) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
               ppart[j+2*nppmx+npoff] = -ppart[j+2*nppmx+npoff];
            }
            if ((dy < edgely) || (dy >= edgery)) {
               dy = ppart[j+nppmx+npoff];
               ppart[j+3*nppmx+npoff] = -ppart[j+3*nppmx+npoff];
            }
         }
/* mixed reflecting/periodic boundary conditions */
         else if (ipbc==3) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
```

```
               ppart[j+2*nppmx+npoff] = -ppart[j+2*nppmx+npoff];
            }
         }
/* set new position */
         ppart[j+npoff] = dx;
         ppart[j+nppmx+npoff] = dy;
         j += blockDim.x;
      }
/* synchronize threads */
      __syncthreads();
/* deposit current to global array */
      nn = nxv - noffp;
      mm = nypmx - moffp;
      nn = mx+1 < nn ? mx+1 : nn;
      mm = my+1 < mm ? my+1 : mm;
      ii = threadIdx.x;
      while (ii < mxv*(my+1)) {
         j = ii/mxv;
         i = ii - mxv*j;
         if ((i < nn) && (j < mm)) {
/* original deposit charge, has data hazard on GPU */
/*          cu[3*(i+noffp+nxv*(j+moffp))] += scu[3*ii];     */
/*          cu[1+3*(i+noffp+nxv*(j+moffp))] += scu[1+3*ii]; */
/*          cu[2+3*(i+noffp+nxv*(j+moffp))] += scu[2+3*ii]; */
/* for devices with compute capability 2.x */
            atomicAdd(&cu[3*(i+noffp+nxv*(j+moffp))],scu[3*ii]);
            atomicAdd(&cu[1+3*(i+noffp+nxv*(j+moffp))],scu[1+3*ii]);
            atomicAdd(&cu[2+3*(i+noffp+nxv*(j+moffp))],scu[2+3*ii]);
         }
         ii += blockDim.x;
      }
   }
   return;
}
```