```
/*--------------------------------------------------------------------*/
__global__ void gpuppgppush2l(float ppart[], float fxy[], int kpic[],
                              int noff, int nyp, float qbm, float dt,
                              float *ek, int nx, int ny, int mx, int my,
                              int idimp, int nppmx, int nxv, int nypmx,
                              int mx1, int mxyp1, int ipbc) {
/* for 2d code, this subroutine updates particle co-ordinates and
   velocities using leap-frog scheme in time and first-order linear
   interpolation in space, with various boundary conditions
   threaded version using guard cells, for distributed data
   data read in tiles
   particles stored segmented array
   42 flops/particle, 12 loads, 4 stores
   input: all except ihole, output: ppart, ek
   equations used are:
   vx(t+dt/2) = vx(t-dt/2) + (q/m)*fx(x(t),y(t))*dt,
   vy(t+dt/2) = vy(t-dt/2) + (q/m)*fy(x(t),y(t))*dt,
   where q/m is charge/mass, and
   x(t+dt) = x(t) + vx(t+dt/2)*dt, y(t+dt) = y(t) + vy(t+dt/2)*dt
   fx(x(t),y(t)) and fy(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))
   fy(x,y) = (1-dy)*((1-dx)*fy(n,m)+dx*fy(n+1,m)) + dy*((1-dx)*fy(n,m+1)
      + dx*fy(n+1,m+1))
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   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] = velocity vx of particle n in partition in tile m
   ppart[m][3][n] = velocity vy 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)
   in other words, fxy are the convolutions of the electric field
   over the 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
   kinetic energy/mass at time t is also calculated, using
   ek = .125*sum((vx(t+dt/2)+vx(t-dt/2))**2+(vy(t+dt/2)+vy(t-dt/2))**2)
   nx/ny = system length in x/y direction
   mx/my = number of grids in sorting cell in x/y
   idimp = size of phase space = 4
   nppmx = maximum number of particles in tile
   nxv = first dimension of field 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 qtm, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy;
   float x, y, dx, dy, vx, vy;
/* The sizes of the shared memory arrays are as follows: */
/* float sfxy[2*(mx+1)*(my+1)], sek[blockDim.x];         */
/* to conserve memory, sek overlaps with sfxy            */
/* and the name sfxy is used instead of sek              */
   extern __shared__ float sfxy[];
   double sum1;
   qtm = qbm*dt;
   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];
      npoff = idimp*nppmx*k;
      mnoff = moffp + noff;
/* 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[2*ii] = fxy[2*(i+noffp+nxv*(j+moffp))];
            sfxy[1+2*ii] = fxy[1+2*(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];
         nn = x;
         y = ppart[j+npoff+nppmx];
         mm = y;
         dxp = x - (float) nn;
```

```
         dyp = y - (float) mm;
         nn = 2*(nn - noffp) + 2*mxv*(mm - mnoff);
         amx = 1.0f - dxp;
         amy = 1.0f - dyp;
/* find acceleration */
         dx = amx*sfxy[nn];
         dy = amx*sfxy[1+nn];
         dx = amy*(dxp*sfxy[2+nn] + dx);
         dy = amy*(dxp*sfxy[3+nn] + dy);
         nn += 2*mxv;
         vx = amx*sfxy[nn];
         vy = amx*sfxy[1+nn];
         dx += dyp*(dxp*sfxy[2+nn] + vx);
         dy += dyp*(dxp*sfxy[3+nn] + vy);
/* new velocity */
         vx = ppart[j+npoff+nppmx*2];
         vy = ppart[j+npoff+nppmx*3];
         dx = vx + qtm*dx;
         dy = vy + qtm*dy;
/* average kinetic energy */
         vx += dx;
         vy += dy;
         sum1 += (double) (vx*vx + vy*vy);
         ppart[j+npoff+nppmx*2] = dx;
         ppart[j+npoff+nppmx*3] = dy;
/* new position */
         dx = x + dx*dt;
         dy = y + dy*dt;
/* reflecting boundary conditions */
         if (ipbc==2) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
               ppart[j+npoff+nppmx*2] = -ppart[j+npoff+nppmx*2];
            }
            if ((dy < edgely) || (dy >= edgery)) {
               dy = ppart[j+npoff+nppmx];
               ppart[j+npoff+nppmx*3] = -ppart[j+npoff+nppmx*3];
            }
         }
/* mixed reflecting/periodic boundary conditions */
         else if (ipbc==3) {
            if ((dx < edgelx) || (dx >= edgerx)) {
               dx = ppart[j+npoff];
               ppart[j+npoff+nppmx*2] = -ppart[j+npoff+nppmx*2];
            }
         }
/* set new position */
         ppart[j+npoff] = dx;
         ppart[j+npoff+nppmx] = 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] = 0.125f*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;
}
```