```
!-----------------------------------------------------------------------
      attributes(global) subroutine gpurbppush23l(ppart,fxy,bxy,kpic,qbm&
     &,dt,dtc,ci,ek,idimp,nppmx,nx,ny,mx,my,nxv,nyv,mx1,mxy1,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
! 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(1) = (1 - (om*dt/2)**2 + 2*(omx*dt/2)**2)/(1 + (om*dt/2)**2)
!    rot(2) = 2*(omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
!    rot(3) = 2*(-omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
!    rot(4) = 2*(-omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
!    rot(5) = (1 - (om*dt/2)**2 + 2*(omy*dt/2)**2)/(1 + (om*dt/2)**2)
!    rot(6) = 2*(omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
!    rot(7) = 2*(omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
!    rot(8) = 2*(-omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
!    rot(9) = (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(n,1,m) = position x of particle n in tile m
! ppart(n,2,m) = position y of particle n in tile m
```

```
! ppart(n,3,m) = x momentum of particle n in tile m
! ppart(n,4,m) = y momentum of particle n in tile m
! ppart(n,5,m) = z momentum of particle n in tile m
! fxy(1,j,k) = x component of force/charge at grid (j,k)
! fxy(2,j,k) = y component of force/charge at grid (j,k)
! fxy(3,j,k) = z component of force/charge at grid (j,k)
! that is, convolution of electric field over particle shape
! bxy(1,j,k) = x component of magnetic field at grid (j,k)
! bxy(2,j,k) = y component of magnetic field at grid (j,k)
! bxy(3,j,k) = z component of magnetic field at grid (j,k)
! that is, the convolution of magnetic field over particle shape
! kpic = number of particles per tile
! 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
! nyv = second dimension of field arrays, must be >= ny+1
! mx1 = (system length in x direction - 1)/mx + 1
! mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
! ipbc = particle boundary condition = (0,1,2,3) =
! (none,2d periodic,2d reflecting,mixed reflecting/periodic)
      implicit none
      integer, value :: idimp, nppmx, nx, ny, mx, my, nxv, nyv
      integer, value :: mx1, mxy1, ipbc
      real, value :: qbm, dt, dtc, ci
      real, dimension(nppmx,idimp,mxy1) :: ppart
      real, dimension(3,nxv,nyv) :: fxy, bxy
      integer, dimension(mxy1) :: kpic
      real, dimension(mxy1) :: ek
! local data
      integer :: noff, moff, npp, mxv
      integer :: i, j, k, ii, nn, mm, nm, b
      real :: qtmh, ci2, edgelx, edgely, edgerx, edgery, dxp, dyp
      real :: amx, amy, dx, dy, dz, ox, oy, oz, acx, acy, acz
      real :: p2, gami, qtmg, dtg, omxt, omyt, omzt, omt, anorm
      real :: rot1, rot2, rot3, rot4, rot5, rot6, rot7, rot8, rot9
      real :: x, y
! The sizes of the shared memory arrays are as follows:
! real sfxy(3*(mx+1)*(my+1)), sbxy(3*(mx+1)*(my+1))
! real sek(blockDim%x)
! to conserve memory, sek overlaps with sfxy and sbxy
! and the name sfxy is used instead of sek
      real, shared, dimension(*) :: sfxy
      double precision :: sum1
      b = 3*(mx+1)*(my+1)
```

```
      qtmh = 0.5*qbm*dt
      ci2 = ci*ci
      sum1 = 0.0d0
! set boundary values
      edgelx = 0.0
      edgely = 0.0
      edgerx = real(nx)
      edgery = real(ny)
      if (ipbc==2) then
         edgelx = 1.0
         edgely = 1.0
         edgerx = real(nx-1)
         edgery = real(ny-1)
      else if (ipbc==3) then
         edgelx = 1.0
         edgerx = real(nx-1)
      endif
      mxv = mx + 1
! k = tile number
      k = blockIdx%x + gridDim%x*(blockIdx%y - 1)
! loop over tiles
      if (k <= mxy1) then
         noff = (k - 1)/mx1
         moff = my*noff
         noff = mx*(k - mx1*noff - 1)
         npp = kpic(k)
! load local fields from global array
         nn = min(mx,nx-noff) + 1
         mm = min(my,ny-moff) + 1
         ii = threadIdx%x
         do while (ii <= mxv*(my+1))
            j = (ii - 1)/mxv
            i = ii - mxv*j
            j = j + 1
            if ((i <= nn) .and. (j <= mm)) then
               sfxy(3*ii-2) = fxy(1,i+noff,j+moff)
               sfxy(3*ii-1) = fxy(2,i+noff,j+moff)
               sfxy(3*ii) = fxy(3,i+noff,j+moff)
            endif
            ii = ii + blockDim%x
         enddo
         ii = threadIdx%x
         do while (ii <= mxv*(my+1))
            j = (ii - 1)/mxv
            i = ii - mxv*j
            j = j + 1
            if ((i <= nn) .and. (j <= mm)) then
               sfxy(b+3*ii-2) = bxy(1,i+noff,j+moff)
               sfxy(b+3*ii-1) = bxy(2,i+noff,j+moff)
               sfxy(b+3*ii) = bxy(3,i+noff,j+moff)
            endif
            ii = ii + blockDim%x
         enddo
! synchronize threads
```

```
         call syncthreads()
! loop over particles in tile
         j = threadIdx%x
         do while (j <= npp)
! find interpolation weights
            x = ppart(j,1,k)
            y = ppart(j,2,k)
            nn = x
            mm = y
            dxp = x - real(nn)
            dyp = y - real(mm)
            nm = 3*(nn - noff) + 3*mxv*(mm - moff) + 1
            amx = 1.0 - dxp
            amy = 1.0 - 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 = nn + 3*mxv
            acx = amx*sfxy(nn)
            acy = amx*sfxy(nn+1)
            acz = amx*sfxy(nn+2)
            mm = nn + 3
            dx = dx + dyp*(dxp*sfxy(mm) + acx)
            dy = dy + dyp*(dxp*sfxy(mm+1) + acy)
            dz = dz + dyp*(dxp*sfxy(mm+2) + acz)
! find magnetic field
            nn = nm + b
            ox = amx*sfxy(nn)
            oy = amx*sfxy(nn+1)
            oz = amx*sfxy(nn+2)
            mm = nn + 3
            ox = amy*(dxp*sfxy(mm) + ox)
            oy = amy*(dxp*sfxy(mm+1) + oy)
            oz = amy*(dxp*sfxy(mm+2) + oz)
            nn = nn + 3*mxv
            acx = amx*sfxy(nn)
            acy = amx*sfxy(nn+1)
            acz = amx*sfxy(nn+2)
            mm = nn + 3
            ox = ox + dyp*(dxp*sfxy(mm) + acx)
            oy = oy + dyp*(dxp*sfxy(mm+1) + acy)
            oz = oz + dyp*(dxp*sfxy(mm+2) + acz)
! calculate half impulse
            dx = qtmh*dx
            dy = qtmh*dy
            dz = qtmh*dz
! half acceleration
            acx = ppart(j,3,k) + dx
```

```
            acy = ppart(j,4,k) + dy
            acz = ppart(j,5,k) + dz
! find inverse gamma
            p2 = acx*acx + acy*acy + acz*acz
            gami = 1.0/sqrt(1.0 + p2*ci2)
! renormalize magnetic field
            qtmg = qtmh*gami
! time-centered kinetic energy
            sum1 = sum1 + gami*p2/(1.0 + gami)
! calculate cyclotron frequency
            omxt = qtmg*ox
            omyt = qtmg*oy
            omzt = qtmg*oz
! calculate rotation matrix
            omt = omxt*omxt + omyt*omyt + omzt*omzt
            anorm = 2.0/(1.0 + omt)
            omt = 0.5*(1.0 - 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 + rot4
            rot3 = -omyt + rot7
            rot7 = omyt + rot7
            rot6 = omxt + rot8
            rot8 = -omxt + rot8
! new momentum
            dx = (rot1*acx + rot2*acy + rot3*acz)*anorm + dx
            dy = (rot4*acx + rot5*acy + rot6*acz)*anorm + dy
            dz = (rot7*acx + rot8*acy + rot9*acz)*anorm + dz
            ppart(j,3,k) = dx
            ppart(j,4,k) = dy
            ppart(j,5,k) = dz
! update inverse gamma
            p2 = dx*dx + dy*dy + dz*dz
            dtg = dtc/sqrt(1.0 + p2*ci2)
! new position
            dx = x + dx*dtg
            dy = y + dy*dtg
! reflecting boundary conditions
            if (ipbc==2) then
               if ((dx < edgelx).or.(dx >= edgerx)) then
                  dx = ppart(j,1,k)
                  ppart(j,3,k) = -ppart(j,3,k)
               endif
               if ((dy < edgely).or.(dy >= edgery)) then
                  dy = ppart(j,2,k)
                  ppart(j,4,k) = -ppart(j,4,k)
               endif
! mixed reflecting/periodic boundary conditions
            else if (ipbc==3) then
```

```
              if ((dx < edgelx).or.(dx >= edgerx)) then
                 dx = ppart(j,1,k)
                 ppart(j,3,k) = -ppart(j,3,k)
              endif
            endif
! set new position
            ppart(j,1,k) = dx
            ppart(j,2,k) = dy
            j = j + blockDim%x
         enddo
! synchronize threads
         call syncthreads()
! add kinetic energies in tile
         sfxy(threadIdx%x) = real(sum1)
! synchronize threads
         call syncthreads()
         call lsum2(sfxy,blockDim%x)
! normalize kinetic energy of tile
         if (threadIdx%x==1) ek(k) = sfxy(1)
      endif
      end subroutine
```

```
!-----------------------------------------------------------------------
      attributes(global) subroutine gpu2ppost2l(ppart,q,kpic,qm,nppmx,  &
     &idimp,mx,my,nxv,nyv,mx1,mxy1)
! for 2d code, this subroutine calculates particle charge density
! using first-order linear interpolation, periodic boundaries
! threaded version using guard cells
! 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(n,1,m) = position x of particle n in tile m
! ppart(n,2,m) = position y of particle n in tile m
! q(j,k) = charge density at grid point j,k
! kpic = number of particles per tile
! qm = charge on particle, in units of e
! nppmx = maximum number of particles in tile
! idimp = size of phase space = 4
! mx/my = number of grids in sorting cell in x/y
! nxv = first dimension of charge array, must be >= nx+1
! nyv = second dimension of charge array, must be >= ny+1
! mx1 = (system length in x direction - 1)/mx + 1
! mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
      implicit none
      integer, value :: nppmx, idimp, mx, my, nxv, nyv, mx1, mxy1
      real, value :: qm
      real, dimension(nppmx,idimp,mxy1) :: ppart
      real, dimension(nxv,nyv) :: q
      integer, dimension(mxy1) :: kpic
! local data
      integer :: noff, moff, npp, mxv
      integer :: i, j, k, ii, nn, mm, np, mp
      real :: dxp, dyp, amx, amy, old
! The size of the shared memory array is as follows:
! real sq((mx+1)*(my+1))
      real, shared, dimension((mx+1)*(my+1)) :: sq
      mxv = mx + 1
! k = tile number
      k = blockIdx%x + gridDim%x*(blockIdx%y - 1)
! loop over tiles
      if (k <= mxy1) then
         noff = (k - 1)/mx1
         moff = my*noff
         noff = mx*(k - mx1*noff - 1)
         npp = kpic(k)
! zero out local accumulator
         i = threadIdx%x
         do while (i <= mxv*(my+1))
            sq(i) = 0.0
```

```
            i = i + blockDim%x
         enddo
! synchronize threads
         call syncthreads()
! loop over particles in tile
         j = threadIdx%x
         do while (j <= npp)
! find interpolation weights
            dxp = ppart(j,1,k)
            nn = dxp
            dyp = ppart(j,2,k)
            mm = dyp
            dxp = qm*(dxp - real(nn))
            dyp = dyp - real(mm)
            nn = nn - noff + 1
            mm = mxv*(mm - moff)
            amx = qm - dxp
            mp = mm + mxv
            amy = 1.0 - dyp
            np = nn + 1
! deposit charge within tile to local accumulator
! original deposit charge, has data hazard on GPU
!           sq(np+mp) = sq(np+mp) + dxp*dyp
!           sq(nn+mp) = sq(nn+mp) + amx*dyp
!           sq(np+mm) = sq(np+mm) + dxp*amy
!           sq(nn+mm) = sq(nn+mm) + amx*amy
! for devices with compute capability 2.x
            old = atomicAdd(sq(np+mp),dxp*dyp)
            old = atomicAdd(sq(nn+mp),amx*dyp)
            old = atomicAdd(sq(np+mm),dxp*amy)
            old = atomicAdd(sq(nn+mm),amx*amy)
            j = j + blockDim%x
         enddo
! synchronize threads
         call syncthreads()
! deposit charge to global array
         nn = min(mxv,nxv-noff)
         mm = min(my+1,nyv-moff)
         ii = threadIdx%x
         do while (ii <= mxv*(my+1))
            j = (ii - 1)/mxv
            i = ii - mxv*j
            j = j + 1
            if ((i <= nn) .and. (j <= mm)) then
! original deposit charge, has data hazard on GPU
!              q(i+noff,j+moff) = q(i+noff,j+moff) + sq(ii)
! for devices with compute capability 2.x
               old = atomicAdd(q(i+noff,j+moff),sq(ii))
            endif
            ii = ii + blockDim%x
         enddo
      endif
      end subroutine
```

```
!-----------------------------------------------------------------------
      attributes(global) subroutine gpu2rjppost2l(ppart,cu,kpic,qm,dt,ci&
     &,nppmx,idimp,nx,ny,mx,my,nxv,nyv,mx1,mxy1,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
! 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(n,1,m) = position x of particle n in tile m
! ppart(n,2,m) = position y of particle n in tile m
! ppart(n,3,m) = x momentum of particle n in tile m
! ppart(n,4,m) = y momentum of particle n in tile m
! ppart(n,5,m) = z momentum of particle n in tile m
! cu(i,j,k) = ith component of current density at grid point j,k
! kpic = number of particles per tile
! 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
! nyv = second dimension of current array, must be >= ny+1
! mx1 = (system length in x direction - 1)/mx + 1
! mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
! ipbc = particle boundary condition = (0,1,2,3) =
! (none,2d periodic,2d reflecting,mixed reflecting/periodic)
      implicit none
      integer, value :: nppmx, idimp, nx, ny, mx, my, nxv, nyv
      integer, value :: mx1, mxy1, ipbc
      real, value :: qm, dt, ci
      real, dimension(nppmx,idimp,mxy1) :: ppart
      real, dimension(3,nxv,nyv) :: cu
      integer, dimension(mxy1) :: kpic
! local data
      integer :: noff, moff, npp, mxv
      integer :: i, j, k, ii, nn, mm, old
      real :: ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy
      real :: x, y, dx, dy, vx, vy, vz, p2, gami
! The size of the shared memory array is as follows:
! real scu(3*(mx+1)*(my+1))
      real, shared, dimension(3*(mx+1)*(my+1)) :: scu
```

```
      ci2 = ci*ci
! set boundary values
      edgelx = 0.0
      edgely = 0.0
      edgerx = real(nx)
      edgery = real(ny)
      if (ipbc==2) then
         edgelx = 1.0
         edgely = 1.0
         edgerx = real(nx-1)
         edgery = real(ny-1)
      else if (ipbc==3) then
         edgelx = 1.0
         edgerx = real(nx-1)
      endif
      mxv = mx + 1
! k = tile number
      k = blockIdx%x + gridDim%x*(blockIdx%y - 1)
! loop over tiles
      if (k <= mxy1) then
         noff = (k - 1)/mx1
         moff = my*noff
         noff = mx*(k - mx1*noff - 1)
         npp = kpic(k)
! zero out local accumulator
         i = threadIdx%x
         do while (i <= 3*mxv*(my+1))
            scu(i) = 0.0
            i = i + blockDim%x
         enddo
! synchronize threads
         call syncthreads()
! loop over particles in tile
         j = threadIdx%x
         do while (j <= npp)
! find interpolation weights
            x = ppart(j,1,k)
            y = ppart(j,2,k)
            nn = x
            mm = y
            dxp = qm*(x - real(nn))
            dyp = y - real(mm)
! find inverse gamma
            vx = ppart(j,3,k)
            vy = ppart(j,4,k)
            vz = ppart(j,5,k)
            p2 = vx*vx + vy*vy + vz*vz
            gami = 1.0/sqrt(1.0 + p2*ci2)
! calculate weights
            nn = 3*(nn - noff) + 3*mxv*(mm - moff) + 1
            amx = qm - dxp
            amy = 1.0 - dyp
! deposit current
            dx = amx*amy
```

```
            dy = dxp*amy
            vx = vx*gami
            vy = vy*gami
            vz = vz*gami
! original current deposit, has data hazard on GPU
!           scu(nn) = scu(nn) + vx*dx
!           scu(nn+1) = scu(nn+1) + vy*dx
!           scu(nn+2) = scu(nn+2) + vz*dx
! for devices with compute capability 2.x
            old = atomicAdd(scu(nn),vx*dx)
            old = atomicAdd(scu(nn+1),vy*dx)
            old = atomicAdd(scu(nn+2),vz*dx)
            dx = amx*dyp
            mm = nn + 3
! original current deposit, has data hazard on GPU
!           scu(mm) = scu(mm) + vx*dy
!           scu(mm+1) = scu(mm+1) + vy*dy
!           scu(mm+2) = scu(mm+2) + vz*dy
! for devices with compute capability 2.x
            old = atomicAdd(scu(mm),vx*dy)
            old = atomicAdd(scu(mm+1),vy*dy)
            old = atomicAdd(scu(mm+2),vz*dy)
            dy = dxp*dyp
            nn = nn + 3*mxv
! original current deposit, has data hazard on GPU
!           scu(nn) = scu(nn) + vx*dx
!           scu(nn+1) = scu(nn+1) + vy*dx
!           scu(nn+2) = scu(nn+2) + vz*dx
! for devices with compute capability 2.x
            old = atomicAdd(scu(nn),vx*dx)
            old = atomicAdd(scu(nn+1),vy*dx)
            old = atomicAdd(scu(nn+2),vz*dx)
            mm = nn + 3
! original current deposit, has data hazard on GPU
!           scu(mm) = scu(mm) + vx*dy
!           scu(mm+1) = scu(mm+1) + vy*dy
!           scu(mm+2) = scu(mm+2) + vz*dy
! for devices with compute capability 2.x
            old = atomicAdd(scu(mm),vx*dy)
            old = atomicAdd(scu(mm+1),vy*dy)
            old = 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) then
               if ((dx < edgelx).or.(dx >= edgerx)) then
                  dx = ppart(j,1,k)
                  ppart(j,3,k) = -ppart(j,3,k)
               endif
               if ((dy < edgely).or.(dy >= edgery)) then
                  dy = ppart(j,2,k)
                  ppart(j,4,k) = -ppart(j,4,k)
               endif
```

```
! mixed reflecting/periodic boundary conditions
            else if (ipbc==3) then
               if ((dx < edgelx).or.(dx >= edgerx)) then
                  dx = ppart(j,1,k)
                  ppart(j,3,k) = -ppart(j,3,k)
               endif
            endif
! set new position
            ppart(j,1,k) = dx
            ppart(j,2,k) = dy
            j = j + blockDim%x
         enddo
! synchronize threads
         call syncthreads()
! deposit current to global array
         nn = min(mxv,nxv-noff)
         mm = min(my+1,nyv-moff)
         ii = threadIdx%x
         do while (ii <= mxv*(my+1))
            j = (ii - 1)/mxv
            i = ii - mxv*j
            j = j + 1
            if ((i <= nn) .and. (j <= mm)) then
! original current deposit, has data hazard on GPU
!              cu(1,i+noff,j+moff) = cu(1,i+noff,j+moff) + scu(3*ii-2)
!              cu(2,i+noff,j+moff) = cu(2,i+noff,j+moff) + scu(3*ii-1)
!              cu(3,i+noff,j+moff) = cu(3,i+noff,j+moff) + scu(3*ii)
! for devices with compute capability 2.x
               old = atomicAdd(cu(1,i+noff,j+moff),scu(3*ii-2))
               old = atomicAdd(cu(2,i+noff,j+moff),scu(3*ii-1))
               old = atomicAdd(cu(3,i+noff,j+moff),scu(3*ii))
            endif
            ii = ii + blockDim%x
         enddo
      endif
      end subroutine
```