

```

!-----
      attributes(global) subroutine gpuppggrbpush231(ppart, fxy, bxy, kplic, &
      &noff, nyp, qbm, dt, dtc, ci, ek, idimp, nppmx, nx, ny, mx, my, nxv, nypmx, mx1, &
      &mxyp1, 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:
!    $rot(1) = (1 - (om*dt/2)**2 + 2*(omx*dt/2)**2)/(1 + (om*dt/2)**2)$ 
!    $rot(2) = 2*(omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(3) = 2*(-omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(4) = 2*(-omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(5) = (1 - (om*dt/2)**2 + 2*(omy*dt/2)**2)/(1 + (om*dt/2)**2)$ 
!    $rot(6) = 2*(omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(7) = 2*(omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(8) = 2*(-omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)$ 
!    $rot(9) = (1 - (om*dt/2)**2 + 2*(omz*dt/2)**2)/(1 + (om*dt/2)**2)$ 
! 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 partition in tile m

```

```

! ppart(n,2,m) = position y of particle n in partition in tile m
! ppart(n,3,m) = momentum vx of particle n in partition in tile m
! ppart(n,4,m) = momentum vy of particle n in partition in tile m
! ppart(n,5,m) = momentum vz of particle n in partition in tile m
! fxy(1,j,k) = x component of force/charge at grid (j,kk)
! fxy(2,j,k) = y component of force/charge at grid (j,kk)
! fxy(3,j,k) = z component of force/charge at grid (j,kk)
! that is, convolution of electric field over particle shape,
! where kk = k + noff - 1
! bxy(1,j,k) = x component of magnetic field at grid (j,kk)
! bxy(2,j,k) = y component of magnetic field at grid (j,kk)
! bxy(3,j,k) = z component of magnetic field at grid (j,kk)
! that is, the convolution of magnetic field over particle shape,
! where kk = k + noff - 1
! 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 = reciprocal 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
! mxypl = mx1*mypl, where mypl=(partition 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 :: noff, nyp, idimp, nppmx, nx, ny, mx, my
! integer, value :: nxv, nypmx, mx1, mxypl, ipbc
! real, value :: qbm, dt, dtc, ci
! real, dimension(nppmx,idimp,mxypl) :: ppart
! real, dimension(3,nxv,nypmx) :: fxy, bxy
! integer, dimension(mxypl) :: kpic
! real, dimension(mxypl) :: ek
! local data
! integer :: noffp, moffp, nppp, mxv
! integer :: mnoff, i, j, k, ii, nn, mm, nm, b
! real :: qtmh, edgelx, edgely, edgerx, edgerly, dxp, dyp, amx, amy
! real :: dx, dy, dz, ox, oy, oz, acx, acy, acz, ci2, p2, gami
! real :: 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 sfx(3*(mx+1)*(my+1)), sbxy(3*(mx+1)*(my+1))
! real sek(blockDim%x)

```

```

! to conserve memory, sek overlaps with sfxxy and sbxy
! and the name sfxxy is used instead of sek
    real, shared, dimension(*) :: sfxxy
    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 = 1.0
    edgerx = real(nx)
    edgery = real(ny-1)
    if ((ipbc.eq.2).or.(ipbc.eq.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 <= mxyp1) then
        noffp = (k - 1)/mx1
        moffp = my*noffp
        noffp = mx*(k - mx1*noffp - 1)
        nppp = kp1c(k)
        mnoff = moffp + noff
! load local fields from global array
        nn = min(mx,nx-noffp) + 1
        mm = min(my,nyp-moffp) + 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
                sfxxy(3*ii-2) = fxy(1,i+noffp,j+moffp)
                sfxxy(3*ii-1) = fxy(2,i+noffp,j+moffp)
                sfxxy(3*ii) = fxy(3,i+noffp,j+moffp)
            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
                sfxxy(b+3*ii-2) = bxy(1,i+noffp,j+moffp)
                sfxxy(b+3*ii-1) = bxy(2,i+noffp,j+moffp)
                sfxxy(b+3*ii) = bxy(3,i+noffp,j+moffp)
            endif
            ii = ii + blockDim%x
        enddo
    endif
enddo

```

```

! synchronize threads
    call syncthread()
! loop over particles in tile
    j = threadIdx%x
    do while (j <= nppp)
! find interpolation weights
    x = ppart(j,1,k)
    nn = x
    y = ppart(j,2,k)
    mm = y
    dxp = x - real(nn)
    dyp = y - real(mm)
    nm = 3*(nn - noffp) + 3*mxv*(mm - mnoff) + 1
    amx = 1.0 - dxp
    amy = 1.0 - dyp
! find electric field
    nn = nm
    dx = amx*sfxxy(nn)
    dy = amx*sfxxy(nn+1)
    dz = amx*sfxxy(nn+2)
    mm = nn + 3
    dx = amy*(dxp*sfxxy(mm) + dx)
    dy = amy*(dxp*sfxxy(mm+1) + dy)
    dz = amy*(dxp*sfxxy(mm+2) + dz)
    nn = nn + 3*mxv
    acx = amx*sfxxy(nn)
    acy = amx*sfxxy(nn+1)
    acz = amx*sfxxy(nn+2)
    mm = nn + 3
    dx = dx + dyp*(dxp*sfxxy(mm) + acx)
    dy = dy + dyp*(dxp*sfxxy(mm+1) + acy)
    dz = dz + dyp*(dxp*sfxxy(mm+2) + acz)
! find magnetic field
    nn = nm + b
    ox = amx*sfxxy(nn)
    oy = amx*sfxxy(nn+1)
    oz = amx*sfxxy(nn+2)
    mm = nn + 3
    ox = amy*(dxp*sfxxy(mm) + ox)
    oy = amy*(dxp*sfxxy(mm+1) + oy)
    oz = amy*(dxp*sfxxy(mm+2) + oz)
    nn = nn + 3*mxv
    acx = amx*sfxxy(nn)
    acy = amx*sfxxy(nn+1)
    acz = amx*sfxxy(nn+2)
    mm = nn + 3
    ox = ox + dyp*(dxp*sfxxy(mm) + acx)
    oy = oy + dyp*(dxp*sfxxy(mm+1) + acy)
    oz = oz + dyp*(dxp*sfxxy(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.eq.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
        endif
! mixed reflecting/periodic boundary conditions

```

```

        else if (ipbc.eq.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 syncthread()
! add kinetic energies in tile
        sfxy(threadIdx%x) = real(sum1)
! synchronize threads
        call syncthread()
        call lsum2(sfxy,blockDim%x)
! normalize kinetic energy of tile
        if (threadIdx%x==1) ek(k) = sfxy(1)
    endif
end subroutine
!

```

```

!-----
      attributes(global) subroutine gpu2ppgppost2l(ppart,q,kpic,noff,qm,&
      &idimp,nppmx,mx,my,nxv,nypmx,mx1,mxypl)
! 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(n,1,m) = position x of particle n in partition in tile m
! ppart(n,2,m) = position y of particle n in partition in tile m
! q(j,k) = charge density at grid point (j,kk),
! where kk = k + noff - 1
! 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
! mxypl = mx1*mypl, where mypl=(partition length in y direction-1)/my+1
      implicit none
      integer, value :: noff, idimp, nppmx, mx, my, nxv, nypmx
      integer, value :: mx1, mxypl
      real, value :: qm
      real, dimension(nppmx,idimp,mxypl) :: ppart
      real, dimension(nxv,nypmx) :: q
      integer, dimension(mxypl) :: kpic
! local data
      integer :: noffp, moffp, nppp, mxv
      integer :: mnoff, i, j, k, ii, nn, np, mm, 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 <= mxypl) then
         noffp = (k - 1)/mx1
         moffp = my*noffp
         noffp = mx*(k - mx1*noffp - 1)
         nppp = kpic(k)
         mnoff = moffp + noff

```

```

! 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 <= nppp)
! 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 - noffp + 1
        mm = mxv*(mm - mnoff)
        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-noffp)
    mm = min(my+1,nypmx-moffp)
    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+noffp,j+moffp) = q(i+noffp,j+moffp) + sq(ii)
! for devices with compute capability 2.x
            old = atomicAdd(q(i+noffp,j+moffp),sq(ii))
        endif
        ii = ii + blockDim%x
    enddo

```



```
        enddo
    endif
end subroutine
!
```

```

!-----
      attributes(global) subroutine gpu2pprjppost2l(ppart,cu,kpic,noff, &
      &qm,dt,ci,nppmx,idimp,nx,ny,mx,my,nxv,nypmx,mx1,mxypl,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(n,1,m) = position x of particle n in partition in tile m
! ppart(n,2,m) = position y of particle n in partition in tile m
! ppart(n,3,m) = x momentum of particle n in partition in tile m
! ppart(n,4,m) = y momentum of particle n in partition in tile m
! ppart(n,5,m) = z momentum of particle n in partition in tile m
! cu(i,j,k) = ith component of current density at grid point (j,kk),
! where kk = k + noff - 1
! 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 = reciprocal 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
! mxypl = mx1*mypl, where mypl=(partition 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 :: noff, nppmx, idimp, nx, ny, mx, my, nxv, nypmx
      integer, value :: mx1, mxypl, ipbc
      real, value :: qm, dt, ci
      real, dimension(nppmx,idimp,mxypl) :: ppart
      real, dimension(3,nxv,nypmx) :: cu
      integer, dimension(mxypl) :: kpic
! local data
      integer :: noffp, moffp, nppp, mxv
      integer :: mnoff, i, j, k, ii, nn, mm
      real :: ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy
      real :: x, y, dx, dy, vx, vy, vz, p2, gami, old
! 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 = 1.0
      edgerx = real(nx)
      edgery = real(ny-1)
      if ((ipbc==2).or.(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 <= mxypl) then
        noffp = (k - 1)/mx1
        moffp = my*noffp
        noffp = mx*(k - mx1*noffp - 1)
        nppp = kpic(k)
        mnoff = moffp + noff
! 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 <= nppp)
! find interpolation weights
          x = ppart(j,1,k)
          nn = x
          y = ppart(j,2,k)
          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 - noffp) + 3*mxv*(mm - mnoff) + 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 = dyp*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.eq.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.eq.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-noffp)
    mm = min(my+1,nypmx-moffp)
    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
!             cu(1,i+noffp,j+moffp) = cu(1,i+noffp,j+moffp) + scu(3*ii-2)
!             cu(2,i+noffp,j+moffp) = cu(2,i+noffp,j+moffp) + scu(3*ii-1)
!             cu(3,i+noffp,j+moffp) = cu(3,i+noffp,j+moffp) + scu(3*ii)
! for devices with compute capability 2.x
            old = atomicAdd(cu(1,i+noffp,j+moffp),scu(3*ii-2))
            old = atomicAdd(cu(2,i+noffp,j+moffp),scu(3*ii-1))
            old = atomicAdd(cu(3,i+noffp,j+moffp),scu(3*ii))
        endif
        ii = ii + blockDim%x
    enddo
endif
end subroutine

```