```
!-----------------------------------------------------------------------
      attributes(global) subroutine gpuppgppush2l(ppart,fxy,kpic,noff,  &
     &nyp,qbm,dt,ek,nx,ny,mx,my,idimp,nppmx,nxv,nypmx,mx1,mxyp1,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, 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(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) = velocity vx of particle n in partition in tile m
! ppart(n,4,m) = velocity vy 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)
! in other words, fxy are the convolutions of the electric field
! over the 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
! 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)
      implicit none
      integer, value :: noff, nyp, nx, ny, mx, my, idimp, nppmx
      integer, value :: nxv, nypmx, mx1, mxyp1, ipbc
      real, value :: qbm, dt
      real, dimension(nppmx,idimp,mxyp1) :: ppart
      real, dimension(2,nxv,nypmx) :: fxy
      integer, dimension(mxyp1) :: kpic
     real, dimension(mxyp1) :: ek
! local data
      integer :: noffp, moffp, nppp, mxv
      integer :: mnoff, i, j, k, ii, nn, mm
      real :: qtm, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy
      real :: x, y, dx, dy, vx, vy
! The sizes of the shared memory arrays are as follows:
! real 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
      real, shared, dimension(*) :: sfxy
      double precision :: sum1
      qtm = qbm*dt
      sum1 = 0.0d0
! 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 <= mxyp1) then
         noffp = (k - 1)/mx1
         moffp = my*noffp
         noffp = mx*(k - mx1*noffp - 1)
         nppp = kpic(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
               sfxy(2*ii-1) = fxy(1,i+noffp,j+moffp)
               sfxy(2*ii) = fxy(2,i+noffp,j+moffp)
            endif
            ii = ii + 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 = x - real(nn)
            dyp = y - real(mm)
            nn = 2*(nn - noffp) + 2*mxv*(mm - mnoff) + 1
            amx = 1.0 - dxp
            amy = 1.0 - dyp
! find acceleration
            dx = amx*sfxy(nn)
            dy = amx*sfxy(nn+1)
            dx = amy*(dxp*sfxy(nn+2) + dx)
            dy = amy*(dxp*sfxy(nn+3) + dy)
            nn = nn + 2*mxv
            vx = amx*sfxy(nn)
            vy = amx*sfxy(nn+1)
            dx = dx + dyp*(dxp*sfxy(nn+2) + vx)
            dy = dy + dyp*(dxp*sfxy(nn+3) + vy)
! new velocity
            vx = ppart(j,3,k)
            vy = ppart(j,4,k)
            dx = vx + qtm*dx
            dy = vy + qtm*dy
! average kinetic energy
            vx = vx + dx
            vy = vy + dy
            sum1 = sum1 + dble(vx*vx + vy*vy)
            ppart(j,3,k) = dx
            ppart(j,4,k) = dy
! new position
            dx = x + dx*dt
            dy = y + dy*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()
! 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) = 0.125*sfxy(1)
      endif
      end subroutine
```

```
!-----------------------------------------------------------------------
      attributes(global) subroutine gpu2ppgppost2l(ppart,q,kpic,noff,qm,&
     &idimp,nppmx,mx,my,nxv,nypmx,mx1,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(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
! mxyp1 = mx1*myp1, where myp1=(partition length in y direction-1)/my+1
      implicit none
      integer, value :: noff, idimp, nppmx, mx, my, nxv, nypmx
      integer, value :: mx1, mxyp1
      real, value :: qm
      real, dimension(nppmx,idimp,mxyp1) :: ppart
      real, dimension(nxv,nypmx) :: q
      integer, dimension(mxyp1) :: 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 <= mxyp1) 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
      endif
      end subroutine
!
```