NVIDIA CUDA Fortran

PAX-HPC CUDA Workshop

Michael Bareford
m.bareford@epcc.ed.ac.uk

Reusing this material

NVIDIA HPC SDK

- Several versions of the NV HPC SDK are installed on Cirrus and accessed via TCL (Tool Command Language) module files.

```
module load nvidia/nvhpc-nompi/24.5
module load openmpi/4.1.6-cuda-12.4-nvfortran
```

- The SDK will contain specific CUDA versions (e.g., 11.8, 12.4) that should be compatible with the underlying GPU driver (v550.144.03).

- SDK contains the `nvfortran` compiler.
  - https://docs.nvidia.com/hpc-sdk/archive/24.5/compilers/hpc-compilers-user-guide/index.html

- NVIDIA CUDA Fortran Programming Guide
  - https://docs.nvidia.com/hpc-sdk/archive/24.5/compilers/cuda-fortran-prog-guide/index.html

- Search archives for docs pertaining to different versions of the SDK.
  - https://docs.nvidia.com/hpc-sdk/archive/index.html

NVIDIA CUDA Fortran

- Show how to offload to GPU two “realistic” Fortran do loops using CUDA Fortran
  - **Loop Alpha** populates an array
    - Each iteration calculates one element
  - **Loop Beta** performs an array reduction
    - Each iteration calculates all elements in an array
    - Iteration arrays are summed

- Highlight the following features of CUDA Fortran (from NVIDIA HPC SDK 24.5, CUDA 12.4)
  - Using the MPI rank to set the GPU device
  - Accessing GPU device capabilities
  - Setting GPU device memory limits
  - Host and Device memory
  - Defining and invoking CUDA kernels
  - Shared memory
  - Streams
  - Kernel loop directives
  - Atomic operations
  - Pinned memory

Compilation

```
module -s load nvidia/nvhpc-nompi/24.5
module -s load openmpi/4.1.6-cuda-24.5-nvfortran


FFLAGS = -I${MPI_HOME}/include -O3 -cpp -r8 \
         -tp=cascadelake -cuda -gpu=cuda12.4,cc70

LIBS = -L${MPI_HOME}/lib -lmpi_mpifh


mpifort --version
    > nvfortran 24.5-0 64-bit target on x86-64 Linux -tp cascadelake
    > NVIDIA Compilers and Tools
    > Copyright (c) 2024, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.

nvfortran -help
man nvfortran
```

Link MPI rank to GPU device

```
program myprog
  use cudafor
  ...
  implicit none
  ...
  include 'mpif.h'
  ...
  call MPI_Init(ierr)
  ...
  ierr = cudaGetDeviceCount(ndevices)
  ierr = cudaSetDevice(MOD(rank_local/ndevices))
  ...
  call MPI_Finalize(ierr)
end program
```

Link MPI rank to GPU device

```
program myprog
  use cudafor
  ...
  implicit none
  ...
  include 'mpif.h'
  ...
  call MPI_Init(ierr)
  ...
  ierr = cudaGetDeviceCount(ndevices)
  ierr = cudaSetDevice(MOD(rank_local/ndevices))
  ...
  call MPI_Finalize(ierr)
end program
```

The cudafor module exists with NV SDK compilers include folder.

```
/work/y07/shared/cirrus-software/nvidia/
    hpcsdk-24.5/Linux_x86_64/24.5/compilers/include/cudafor.mod
```

Link MPI rank to GPU device

```
program myprog
  use cudafor
  ...
  implicit none
  ...
  include 'mpif.h'
  ...
  call MPI_I
  ...
  ierr = cud
  ierr = cud
  ...
  call MPI_F
end program
```

```
...

# determine rank_local
# node_num() uses MPI_Get_processor_name() to convert compute node name to number

call MPI_Comm_split(MPI_COMM_WORLD, node_num(), rank, mpi_comm_local, ierr)

call MPI_Comm_size(mpi_comm_local, nranks_local, ierr)

rank_local = MOD(rank, nranks_local)

...
```

Accessing GPU Device Capabilities

```
program myprog
  use cudafor
  ...
  type(cudaDeviceProp) :: cuda_prop
  ...

  device_id = rank_local/ndevices
  ...
  ierr = cudaGetDeviceProperties(cuda_prop, device_id)

  ...
end program
```

Accessing GPU Device Capabilities

```
program myprog
  use cudafor
  ...
  type(cudaDeviceProp) :: cuda_prop
  ...

  device_id = rank_local/nc
  ...
  ierr = cudaGetDevicePrope

  ...
end program
```

```
cuda_prop%totalGlobalMem

         %managedMemory
         %unifiedAddressing

         %multiProcessorCount
         %maxBlocksPerMultiProcessor
         %maxThreadsPerBlock
         %regsPerBlock

         %warpsize
```

https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html

Setting GPU Device Memory Limits

`cudaLimitMallocHeapSize` is the size in bytes of the heap used by the `malloc` and `free` device system calls.

```
program myprog
  use cudafor
  ...
  integer(kind=cuda_count_kind) :: cuda_count
  ...

  # Set CUDA malloc heap size to 1 GiB
  cuda_count = 1024*(1024**2)
  ierr = cudaDeviceSetLimit(cudaLimitMallocHeapSize, cuda_count)

  ...
end program
```

Other Limits can be set (apologies for the obscure link).

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__TYPES.html#group__CUDART__TYPES_1g4c4b34c054d383b0e9a63ab0ffc93651

Introducing Loop Alpha...

```
...
real*8 ppos(8,nlocpts) # ppos(:) initialised then never altered
real*8 bc(ncoeffs)     # bc(:) updated before every loop invocation
real*8 xyzf(nlocpts)
...
# ppos() and bc(:) are only read within loop
do i=1,nlocpts

  p1 = ppos(1,i)*D2R
  p2 = ppos(2,i)*D2R
  ra = RAG / ppos(3,i)

  bex = ppos(5,i)
  bey = ppos(6,i)
  bez = ppos(7,i)

  xyzf(i) = XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez, bc)

enddo
```

Introducing Loop Alpha…

```
...
real*8 ppos(8,nlocpts) # ppos(:) initialised then never altered
real*8 bc(ncoeffs)     # bc(:) updated before every loop invocation
real*8 xyzf(nlocpts)
...
# ppos() and bc(:) are only
do i=1,nlocpts

  p1 = ppos(1,i)*D2R
  p2 = ppos(2,i)*D2R
  ra = RAG / ppos(3,i)

  bex = ppos(5,i)
  bey = ppos(6,i)
  bez = ppos(7,i)

  xyzf(i) = XYZsph_alpha(sh

enddo
```

```
real*8 function XYZsph_alpha(shdeg, p1, p2, ra,
                             bex, bey, bez, bc)
  ...
  integer shdeg
  real*8  p1, p2, ra
  real*8  bex, bey, bez, bc(*)
  ...
  XYZsph_alpha = 0.0d0
  ...
  call mk_lf_dlf(...)
  ...
  do il=1,shdeg
    ...
    XYZsph_alpha = XYZsph_alpha + ... + bc(il)
  enddo
  ...
end function XYZsph_alpha
```

Host and Device Memory

```
program myprog
  ...

  real*8, allocatable :: ppos(:,:) # initialised once, read only from then on
  real*8, allocatable :: bc(:)     # updated each time before entering Loop Alpha
  real*8, allocatable :: xyzf(:)   # populated by Loop Alpha

  ...
end program
```

Host and Device Memory

```
program myprog
  ...

  real*8, allocatable :: ppos(:,:) # initialised once, read only from then on
  real*8, allocatable :: bc(:)     # updated each time before entering Loop Alpha
  real*8, allocatable :: xyzf(:)   # populated by Loop Alpha

  ...
end program
```

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:)
  real(8), allocatable, device :: d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...
end module kernels
```

Host and Device Memory

```
program myprog
  ...

  real*8, allocatable :: ppos(:,:) # initialised once, read only from then on
  real*8, allocatable :: bc(:)     # updated each time before entering Loop Alpha
  real*8, allocatable :: xyzf(:)   # populated by Loop Alpha

  ...
end program
```

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:)
  real(8), allocatable, device :: d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)
```

Note the change from `real*8` to `real(8)`.

Other types made to be altered too...
https://docs.nvidia.com/hpc-sdk/archive/24.5/compilers/cuda-fortran-prog-guide/index.html#cfref-dev-code-datatypes

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  attributes(host) subroutine allocate_device_arrays(nd, nlocpts, ncoeffs, ...)
    ...
    allocate(d_ppos(nd,nlocpts), d_bc(ncoeffs))
    allocate(d_xyzf(nlocpts))
    ...
  end subroutine allocate_device_arrays

  ...
end module kernels
```

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  attributes(host) subroutine init_device_arrays(nd, nlocpts, ..., ppos, ...)
    ...
    integer nd, nlocpts
    real(8) ppos(nd,nlocpts)
    ...
    d_ppos = ppos
    ...
  end subroutine init_device_arrays

  ...
end module kernels
```

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  attributes(host) subroutine init_device_arrays(nd, nlocpts, ..., ppos, ...)
    ...
    integer nd, nlocpts
    real(8) ppos(nd,nlocpts)
    ...
    d_ppos = ppos
    ...
  end subroutine init_device_arrays

  ...
end module kernels
```

Avoid using wild character (*) for array dimension sizes.

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  attributes(host) subroutine init_alpha_device_arrays(ncoeffs, ..., bc, ...)
    ...
    integer ncoeffs
    real(8) bc(ncoeffs)
    ...
    d_bc = bc
    ...
  end subroutine init_alpha_device_arrays

  ...
end module kernels
```

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  attributes(host) subroutine get_alpha_device_arrays(nlocpts, ..., xyzf, ...)
    ...
    integer nlocpts
    real(8) xyzf(nlocpts)
    ...
    xyzf = d_xyzf
    ...
  end subroutine get_alpha_device_arrays

  ...
end module kernels
```

Host and Device Memory

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:)

  ...

  public ::
    allocate_device_arrays,
    init_device_arrays,
    init_alpha_device_arrays,
    get_alpha_device_arrays,
    deallocate_device_arrays,

  ...
end module kernels
```

Defining a CUDA Kernel for Loop Alpha

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    d_xyzf(i) = XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez)

  endif
  ...
end subroutine kernel_alpha
```

Defining a CUDA Kernel for Loop Alpha

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    d_xyzf(i) = XYZsph_alpha(shdeg, p1,

  endif
  ...
end subroutine kernel_alpha
```

Block index: x, y, z
Block dimension: x, y, z
Thread index: x, y, z

The 3D block structure defines a grid.
Example here uses just one dimension.

One block per GPU streaming multiprocessor (SM).
Many threads per block (e.g., 128).
Each thread assigned to one SM core.

Defining a CUDA Kernel for Loop Alpha

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .l

    p1 = c
    p2 = c
    ra = R

    bex =
    bey =
    bez =

    d_xyzf

  endif
  ...
end subrou
```

```
attributes(device) real(8) function XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez)
  ...
  integer, value :: shdeg
  real(8), value :: p1, p2, ra
  real(8), value :: bex, bey, bez
  ...
  XYZsph_alpha = 0.0d0
  ...
  call mk_lf_dlf(...)
  ...
  do il=1,shdeg
    ...
    XYZsph_alpha = XYZsph_alpha + ... + d_bc(il)
  enddo
  ...
end function XYZsph_alpha
```

Defining a CUDA Kernel for Loop Alpha

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .l

    p1 = c
    p2 = c
    ra = F

    bex =
    bey =
    bez =

    d_xyzf

  endif
  ...
end subrou
```

```
attributes(device) real(8) function XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez)
  ...
  integer, value :: shdeg
  real(8), value :: p1, p2, ra
  real(8), value :: bex, bey, bez
  ...
  XYZsph_alpha = 0.0d0
  ...
  call mk_lf_dlf(...)
  ...
  do il=1,shdeg
    ...
    XYZsph_alpha = XYZsph_alpha + ... + d_bc(il)
  enddo
  ...
end function XYZsph_alpha
```

```
attributes(device) subroutine mk_lf_dlf(...)
  ...
end subroutine mk_lf_dlf
```

Defining a CUDA Kernel for Loop Alpha

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), d_bc(:)
  real(8), allocatable, device :: d_xyzf(:), d_be(:)

  ...

  private ::
    XYZsph_alpha,
    XYZsph_beta,
    mk_lf_dlf

  public ::
    ...
    kernel_alpha,
    kernel_beta
    ...
  ...
end module kernels
```

All functions/subroutines (global and device) defined within `kernels` module.

Invoking CUDA Kernel for Loop Alpha

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call init_alpha_device_arrays(ncoeffs, ..., bc, ...)

  call kernel_alpha <<< nblocks, nthreads >>> (shdeg, ..., nlocpts, ...)

  istat = cudaDeviceSynchronize()

  call get_alpha_device_arrays(nlocpts, ..., xyzf, ...)

  ...

end subroutine alpha
```

Invoking CUDA Kernel for Loop Alpha

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call init_alpha_device_arrays(ncoeffs, ..., bc, ...)

  call kernel_alpha <<< nblocks, nthreads >>> (shdeg, ..., nlocpts, ...)

  istat = cudaDeviceSynchronize()

  call get_alpha_device_arrays(nlocpt

  ...

end subroutine alpha
```

```
integer(dim3) :: grid, block
...
 nblocksPerDevice = grid%x*grid%y*grid%z
nthreadsPerBlocks = block%x*block%y*block%z
...
call kernel_alpha <<< grid, block >>> (...)
```

Invoking CUDA Kernel for Loop Alpha

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call init_alpha_device_arrays(ncoeffs, ..., bc, ...)

  call kernel_alpha <<< nblocks, nthreads, nbytes_shared, stream_id >>>
      (shdeg, ..., nlocpts, ...)

  istat = cudaDeviceSynchronize()

  call get_alpha_device_arrays(nlocpts, ..., xyzf, ...)

  ...

end subroutine alpha
```

There two optional arguments.

- the number of bytes of memory shared between threads of one block
- the stream id

CUDA Kernel: sharing data

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
  ...
  integer, value :: sdoff
  real(8), shared :: s_data(nbytes_shared)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then
    ...
    sdoff = (shdeg+1)*(threadidx%x-1)
    ...
    bx = s_data(sdoff+ik)*dr
    ...
    d_xyzf(i) = XYZsph_alpha(shdeg, sdoff, ..., bex, bey, bez)
  endif
  ...
end subroutine kernel_alpha
```

CUDA Kernel: sharing data

```
attributes(global) subroutine kernel_alpha(shdeg, nlocpts)
   ...
   integer, value :: sdoff
   real(8), shared :: s_data(nbytes_shared)
   ...
   i = (blockidx%x-1)* blockdim%x + threadidx%x

   if (i .le. nlocpts) then
      ...
      sdoff = (shdeg+1)*(threadidx%x-1)
      ...
      bx = s_data(sdoff+ik)*dr
      ...
      d_xyzf(i) = XYZsph_alpha(shdeg, sdoff, ..., bex, bey, bez)
   endif
   ...
end subroutine kernel_alpha
```

The `nbytes_shared` variable is declared and initialised (via subroutine) within kernels module.

CUDA Kernel: streams

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call init_alpha_device_arrays(ncoeffs, ..., bc, ...)

  call kernel_alpha <<< nblocks, nthreads, 0, stream_alpha >>> (shdeg, ..., nlocpts, ...)
  call kernel_beta <<< nblocks, nthreads, 0, stream_beta >>> (shdeg, ..., nlocpts, ...)

  istat = cudaStreamSynchronize(stream_alpha)

  call get_alpha_device_arrays(nlocpts, ..., xyzf, ...)

  ...

end subroutine alpha
```

CUDA Kernel: streams

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call init_alpha_device_arrays(nco

  call kernel_alpha <<< nblocks, nt
  call kernel_beta <<< nblocks, nth

  istat = cudaStreamSynchronize(str

  call get_alpha_device_arrays(nloc

  ...

end subroutine alpha
```

```
use cudafor
...
integer istat, stream_alpha
...
istat = cudaStreamCreate(stream_alpha)
...
...
istat = cudaStreamDestroy(stream_alpha)
...
```

CUDA Kernel Loop Directive

```
attributes(host) subroutine kernel_loop_alpha(shdeg, nlocpts)
  use cudafor
  ...
!$cuf kernel do <<< *, * >>>
  do i=1,nlocpts

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    d_xyzf(i) = XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez)

  enddo
  ...
end subroutine kernel_loop_alpha
```

CUDA Kernel Loop Directive

```
attributes(host) subroutine kernel_loop_alpha(shdeg, nlocpts)
  use cudafor
  ...
!$cuf kernel do <<< nblocks, nthreads, nbytes_shared, stream_id >>>
  do i=1,nlocpts

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    d_xyzf(i) = XYZsph_alpha(shdeg, p1, p2, ra, bex, bey, bez)

  enddo
  ...
end subroutine kernel_loop_alpha
```

CUDA Kernel Loop Directive

```
attributes(host) subroutine kernel_loop_alpha(shdeg, nlocpts)
  use cudafor
  ...
!$cuf kernel do <<< nblocks, nthreads, nbytes_shared, stream_id >>>
  do i=1,nlocpts

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    d_xyzf(i) = XYZsph_alpha(shdeg, p1,

  enddo
  ...
end subroutine kernel_loop_alpha
```

```
module kernels
  ...

  public ::
    kernel_loop_alpha,
    kernel_loop_beta,
    ...
  ...
end module kernels
```

Check for CUDA Error

```
subroutine alpha(...)
  use cudafor
  use kernels

  ...

  call kernel_alpha <<< nblocks, nthreads >>> (shdeg, ..., nlocpts, ...)

  istat = cudaDeviceSynchronize()

  ierr = cudaGetLastError()
  if (ierr .gt. 0) then
    write(*,*) 'kernel alpha failure: ', ierr,
               ', ', cudaGetErrorString(ierr)
  endif

  ...

end subroutine alpha
```

Introducing Loop Beta...

```
...
real*8 ppos(8,nlocpts) # ppos(:) initialised then never altered
real*8 be(ncoeffs)     # be(:) initialised to zero before entering loop
...
# ppos(:) is read within loop but be(:) is read and written
do i=1,nlocpts

  p1 = ppos(1,i)*D2R
  p2 = ppos(2,i)*D2R
  ra = RAG / ppos(3,i)

  bex = ppos(5,i)
  bey = ppos(6,i)
  bez = ppos(7,i)

  call XYZsph_beta(shdeg, p1, p2, ra, bex, bey, bez, be)

enddo
```

Introducing Loop Beta…

```
...
real*8 ppos(8,nlocpts) # ppos(:) initialised then never altered
real*8 be(ncoeffs)     # be(:) initialised to zero before entering loop
...
# ppos(:) is read within loop but b
do i=1,nlocpts

  p1 = ppos(1,i)*D2R
  p2 = ppos(2,i)*D2R
  ra = RAG / ppos(3,i)

  bex = ppos(5,i)
  bey = ppos(6,i)
  bez = ppos(7,i)

  call XYZsph_beta(shdeg, p1, p2, r

enddo
```

```
subroutine XYZsph_beta(shdeg, p1, p2, ra,
                       bex, bey, bez, be)
  ...
  real*8  bex, bey, bez, be(*)
  real*8, allocatable :: dlf(:), ddlf(:)
  ...
  allocate(dlf(shdeg+1), ddlf(shdeg+1))
  ...
  call mk_lf_dlf(...)
  do il=1,shdeg
    ...
    be(nu)= be(nu) + bex*bx + bey*by + bex*bz
  enddo
  ...
  deallocate(dlf, ddlf)
  ...
end subroutine XYZsph_beta
```

Defining a CUDA Kernel for Loop Beta

```
attributes(global) subroutine kernel_beta(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    call XYZsph_beta(shdeg, p1, p2, ra, bex, bey, bez)

  endif
  ...
end subroutine kernel_alpha
```

Defining a CUDA Kernel for Loop Beta

```
attributes(global) subroutine kernel_beta(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + thr

  if (i .le. nlocpts) then

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    call XYZsph_beta(shdeg, p1, p2, ra

  endif
  ...
end subroutine kernel_alpha
```

```
module kernels
  ...

  real(8), allocatable, device :: d_ppos(:,:), ...
  real(8), allocatable, device :: d_be(:)


  ...

  public ::
    allocate_device_arrays,
    init_device_arrays,
    get_beta_device_arrays,
    deallocate_device_arrays,
    ...
    kernel_beta,
    ...

  ...
end module kernels
```

Defining a CUDA Kernel for Loop Beta

```
attributes(global) subroutine kernel_beta(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then

    p1 = d_ppos(1,i)*D2R
    p2 = d_ppos(2,i)*D2R
    ra = RAG / d_ppos(3,i)

    bex = d_ppos(5,i)
    bey = d_ppos(6,i)
    bez = d_ppos(7,i)

    call XYZsph_beta(shdeg, p1, p2, ra, bex, bey, bez)

  endif
  ...
end subroutine kernel_alpha
```

Defining a CUDA Kernel for Loop Beta

```
attributes(global) subroutine kernel_beta(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocp

    p1 = d_ppos(1,
    p2 = d_ppos(2,
    ra = RAG / d_p

    bex = d_ppos(
    bey = d_ppos(
    bez = d_ppos(

    call XYZsph_be

  endif
  ...
end subroutine ke
```

```
attributes(device) subroutine XYZsph_beta(shdeg, p1, p2, ra,
                                          bex, bey, bez)

  use cudafor
  ...
  real(8) bex, bey, bez
  real(8) dlf(shdeg+1), ddlf(shdeg+1)
  ...
  call mk_lf_dlf(...)
  ...
  do il=1,shdeg
    ...
    istat = atomicadd(d_be(nu), bex*bx + bey*by + bex*bz)
  enddo
  ...
end subroutine XYZsph_beta
```

Defining a CUDA Kernel for Loop Beta

```
attributes(global) subroutine kernel_beta(shdeg, nlocpts)
  ...
  i = (blockidx%x-1)* blockdim%x + threadidx%x

  if (i .le. nlocpts) then

    p1 = d_ppos(1,
    p2 = d_ppos(2,
    ra = RAG / d_r

    call XYZsph_be

  endif
  ...
end subroutine ke
```

```
attributes(device) subroutine XYZsph_beta(shdeg, p1, p2, ra,
                                           bex, bey, bez)

  use cudafor
  ...
```

Many other atomic functions

https://docs.nvidia.com/hpc-sdk/archive/24.5/compilers/cuda-fortran-prog-guide/index.html#cfref-dev-code-atomic-funcs

```
  do il=1,shdeg
    ...
    istat = atomicadd(d_be(nu), bex*bx + bey*by + bex*bz)
  enddo
  ...
end subroutine XYZsph_beta
```

Invoking CUDA Kernel for Loop Beta

```
subroutine beta(...)
  use cudafor
  use kernels

  ...

  call kernel_beta <<< nthreads, nblocks >>> (shdeg, nlocpts)

  istat = cudaDeviceSynchronize()

  call get_beta_device_arrays(ncoeffs, ..., be, ...)

  ...

end subroutine beta
```

Pinned Memory

- Memory allocations on the host are pageable by default.
- The GPU cannot access host pageable memory directly.
- For this reason, the CUDA driver must do the following when transferring data between host and device.
    1. Allocate a temporary page-locked (pinned) array on the host
    2. Copy data from the pageable host array to the pinned host array
    3. Transfer the data from the pinned host array to an array on the device
- Steps 1 and 2 can be skipped however if arrays on the host are declared as "pinned".
- See https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

Pinned Memory

```
program myprog
  ...

  real*8, allocatable :: ppos(:,:) # initialised once, read only from then on
  real*8, allocatable :: bc(:)     # updated each time before entering Loop Alpha
  real*8, allocatable :: xyzf(:)   # populated by Loop Alpha
  real*8, allocatable :: be(:)     # populated by Loop Beta

  ...
end program
```

Pinned Memory

```
program myprog
  ...

  real*8, allocatable, pinned :: ppos(:,:) # initialised once, read only from then on
  real*8, allocatable, pinned :: bc(:)     # updated each time before entering Loop Alpha
  real*8, allocatable, pinned :: xyzf(:)   # populated by Loop Alpha
  real*8, allocatable, pinned :: be(:)     # populated by Loop Beta
  ...
  logical is_pinned

  ...
  allocate(ppos(nd,nlocpts), PINNED=is_pinned)
  if (.not. is_pinned) then
    write(*,*) rank, ': ppos not pinned!'
  endif
  ...

end program
```