#6 Using Texture Cache

Key concepts

On-chip read-only data cache accessible through texture pointers or implicit generation of LDG instructions for intent(in) arguements.

Texture pointers

- Device variables using the “texture” and “pointer” attributes
- Declared at module scope
- Texture “binding” and “unbinding” using pointer syntax

LDG instruction

- **ldg()** instruction loads data through texture path
- Generated implicitly for intent(in) array arguments
- For devices of compute capability 3.5 and higher
- Verify using **-Mcuda=keepptx** and look for **ld.global.nc***

```
module kernels
  real, pointer, texture :: bTex(:)
  contains
    attributes(global) subroutine addTex(a,n)
    real :: a(*)
    integer, value :: n
    integer :: i
    i=(blockIdx%x-1)*blockDim%x+threadIdx%x
    if (i <= n) a(i) = a(i)+bTex(i)
  end subroutine addTex

  attributes(global) subroutine addLDG(a,b,n)
    implicit none
    real :: a(*)
    real, intent(in) :: b(*)
    integer, value :: n
    integer :: i
    i=(blockIdx%x-1)*blockDim%x+threadIdx%x
    if (i <= n) a(i) = a(i)+b(i)
  end subroutine addLDG
end module kernels

program texLDG
  use kernels
  integer, parameter :: nb=1000, nt=256
  integer, parameter :: n = nb*nt
  real, device :: a_d(n)
  real, device, target :: b_d(n)
  real :: a(n)

  a_d = 1.0; b_d = 1.0
  bTex => b_d  ! "bind" texture to b_d
  call addTex<<<nb,nt>>>(a_d,n)
  a = a_d
  if (all(a == 2.0)) print *, "texture OK"
  nullify(bTex) ! unbind texture

  call addLDG<<<nb,nt>>>(a_d, b_d, n)
  a = a_d
  if (all(a == 3.0)) print *, "LDG OK"
end program texLDG
```

#7 OpenACC Interoperability

Key concepts

- Using CUDA Fortran device and managed data in OpenACC compute constructs
- Calling CUDA Fortran kernels using OpenACC data present in device memory
- CUDA Fortran and OpenACC sharing CUDA streams
- CUDA Fortran and OpenACC both using multiple devices
- Interoperability with OpenMP programs
- Compiler options: **-Mcuda -ta=tesla -mp**

```
module kernel
  integer, parameter :: n = 256
contains
  attributes(global) subroutine add(a,b)
    integer a(*), b(*)
    i = threadIdx%x
    if(i<=n)a(i)=a(i)+b(i)
  end subroutine add
end module kernel

subroutine interop(me)
use cudafor; use openacc; use kernel
integer, allocatable :: a(:)
integer, device, allocatable :: b(:)
integer(kind=cuda_stream_kind) istr, isync
istat = cudaSetDevice(me)
allocate(a(n), b(n))
a = 1
call acc_set_device_num(me,acc_device_nvidia)
!$acc data copy(a)
isync = me+1
!!! CUDA Device arrays in Accelerator regions
!$acc kernels loop async(isync)
do i=1,n
   b(i) = me
end do
!!! CUDA Fortran kernels using OpenACC data
!$acc host_data use_device(a)
istr = acc_get_cuda_stream(isync)
call add<<<1,n,0,istr>>>(a,b)
!$acc end host_data
!$acc wait
!$acc end data
if (all(a.eq.me+1)) print *,me," PASSED"
end subroutine interop

program testInterop
  use omp_lib
  !$omp parallel
  call interop(omp_get_thread_num())
  !$omp end parallel
end program testInterop
```

CUDA Fortran

Porting Guide

CUDA Fortran is the Fortran analog of the NVIDIA CUDA C language for programming GPUs. This guide includes examples of common language features used when porting Fortran applications to GPUs. These examples cover basic concepts, such as allocating data in host and device memories, transferring data between these spaces, and writing routines that execute on the GPU. Additional topics include using managed memory (memory shared between the host and device), interfacing with CUDA libraries, performing reductions on the GPU, the basics of multi-GPU programming, using the on-chip, read-only texture cache, and interoperability between CUDA Fortran and OpenACC.

See also the CUDA Fortran Quick Reference Card.

Examples used in this guide are available for download at: http://www.pgroup.com/lit/samples/cufport.tar

PGI Compilers & Tools

www.pgroup.com/cudafortran

#1 Simple Increment Code

Key concepts

Host—CPU and its memory

- The **cudafor** module incudes CUDA Fortran definitions and the runtime API
- The **device** variable attribute denotes data which reside in device memory
- Data transfers between host and device are performed with assignment statements (=)
- The *execution configuration*, specified between the triple chevrons <<< … >>>, denotes the number of threads with which a kernel is launched

Device—GPU and its memory

- **attributes(global)** subroutines, or kernels, run on the device by many threads in parallel
- The **value** variable attribute is used for pass-by-value scalar host arguments
- The predefined variables **blockDim**, **blockIdx**, and **threadIdx** are used to enumerate the threads executing the kernel

```
module simpleOps_m
  integer, parameter :: n = 1024*1024
contains
  attributes(global) subroutine inc(a, b)
    integer, intent(inout) :: a(*)
    integer, value :: b
    integer :: i
    i=blockDim%x*(blockIdx%x-1)+threadIdx%x
    if (i <= n) a(i) = a(i) + b
  end subroutine inc
end module simpleOps_m

program incTest
  use cudafor
  use simpleOps_m
  integer, allocatable :: a(:)
  integer, device, allocatable :: a_d(:)
  integer :: b, tPB = 256

  allocate(a(n), a_d(n))
  a = 1; b = 3

  a_d = a
  call inc<<<((N+tPB-1)/tPB),tPB>>> (a_d, b)
  a = a_d

  if (all(a == 4)) print *, "Test Passed"
  deallocate(a, a_d)
end program incTest
```

#2 Using CUDA Managed Data

Key concepts

- Managed Data is a single variable declaration that can be used in both host and device code
- Host code uses the managed variable attribute
- Variables declared with managed attribute require no explicit data transfers between host and device
- Device code is unchanged
- After kernel launch, synchronize before using data on host
- Requires cc30 or higher, cuda6.0 or later

```
module kernels
  integer, parameter :: n = 32
contains
  attributes(global) subroutine increment(a)
    integer :: a(:), i
    i=(blockIdx%x-1)*blockDim%x+threadIdx%x
    if (i <= n) a(i) = a(i)+1
  end subroutine increment
end module kernels

program testManaged
  use kernels
  integer, managed :: a(n)
  a = 4
  call increment<<<1,n>>>(a)
  istat = cudaDeviceSynchronize()
  if (all(a==5)) write(*,*) 'Test Passed'
end program testManaged
```

Managed Data and Derived Types

- Managed attribute applies to all static members, recursively
- Deep copies avoided, only necessary members are transferred between host and device

```
program testManaged
  integer, parameter :: n = 1024
  type ker
     integer :: idec
  end type ker
  integer, managed :: a(n)
  type(ker), managed :: k(n)
  a(:)=2; k(:)%idec=1
  !$cuf kernel do <<<*,*>>>
  do i = 1, n
     a(i) = a(i) - k(i)%idec
  end do
  i = cudaDeviceSynchronize()
  if (all(a==1)) print *, 'Test Passed'
end program testManaged
```

#3 Calling CUDA Libraries

Key concepts

- CUBLAS is the CUDA implementation of the BLAS library
- Interfaces and definition in the cublas module
- Overloaded functions take device and managed array arguments
- Compile with **-lcublas -lblas** for device and host BLAS routines
- User written interfaces to other C libraries

```
program sgemmDevice
  use cudafor
  use cublas
  interface sort
    subroutine sort_int(array, n) &
    bind(C,name='thrust_float_sort_wrapper')
    real, device, dimension(*) :: array
    integer, value :: n
    end subroutine sort_int
  end interface sort
  integer, parameter :: m = 100, n=m, k=m
  real :: a(m,k), b(k,n)
  real, managed :: c(m,n)
  real, device :: a_d(m,k),b_d(k,n),c_d(m,n)
  real, parameter :: alpha = 1.0, beta = 0.0
  integer :: lda = m, ldb = k, ldc = m
  integer :: istat

  a = 1.0; b = 2.0; c = 0.0
  a_d = a; b_d = b

  istat = cublasInit()

  ! Call using cublas names
  call cublasSgemm('n','n',m,n,k, &
       alpha,a_d,lda,b_d,ldb,beta,c,ldc)
  istat = cudaDeviceSynchronize()
  print *, "Max error =", maxval(c)-k*2.0

  ! Overloaded blas name using device data
  call Sgemm('n','n',m,n,k, &
       alpha,a_d,lda,b_d,ldb,beta,c_d,ldc)
  c = c_d
  print *, "Max error =", maxval(c-k*2.0)

  ! Host blas routine using host data
  call random_number(b)
  call Sgemm('n','n',m,n,k, &
       alpha,a,lda,b,ldb,beta,c,ldc)

  ! Sort the results on the device
  call sort(c, m*n)
  print *,c(1,1),c(m,n)
end program sgemmDevice
```

#4 Handling Reductions

Key concepts

- F90 intrinsics (sum, maxval, minval) overloaded to operate on device data; can operate on subarrays
- Optional supported arguments **dim** and **mask** (if managed)
- Interfaces included in **cudafor** module
- CUF kernels for more complicated reductions; can take execution configurations with wildcards
- Requires cc30 or higher, CUDA 6.0 or later

```
program testReduction
  use cudafor
  integer, parameter :: n = 1000
  integer :: a(n), i, res
  integer, device :: a_d(n), b_d(n)

  a(:) = 1
  a_d = a
  b_d = 2
  if (sum(a) == sum(a_d)) &
     print *, "Intrinsic Test Passed"

  res = 0
  !$cuf kernel do <<<*,*>>>
  do i = 1, n
     res = res + a_d(i) * b_d(i)
  enddo
  if (sum(a)*2 == res) &
     print *, "CUF kernel Test Passed"

  if (sum(b_d(1:n:2)) == sum(a_d)) &
     print *, "Subarray Test Passed"

  call multidimred()

end program testReduction

subroutine multidimred()
  use cudafor
  real(8), managed :: a(5,5,5,5,5)
  real(8), managed :: b(5,5,5,5)
  real(8) :: c
  call random_number(a)
  do idim = 1, 5
    b = sum(a,dim=idim)
    c = max(maxval(b),c)
  end do
  print *,"Max Along Any Dimension",c
end subroutine multdimred
```

#5 Running with Multiple GPUs

Key concepts

- **cudaSetDevice()** sets current device
- Operations occur on current device
- Use allocatable data with allocation after **cudaSetDevice** to get data on respective devices
- Kernel code remains unchanged, only host code is modified

```
module kernel
contains
  attributes(global) subroutine assign(a, v)
    implicit none
    real :: a(*)
    real, value :: v
    a(threadIdx%x) = v
  end subroutine assign
end module kernel

program minimal
  use cudafor
  use kernel
  implicit none
  integer, parameter :: n=32
  real :: a(n)
  real, device, allocatable :: a0_d(:)
  real, device, allocatable :: a1_d(:)
  integer :: nDevices, istat

  istat = cudaGetDeviceCount(nDevices)
  if (nDevices < 2) then
     print *, "This program requires ≥2 GPUs"
     stop
  end if

  istat = cudaSetDevice(0)
  allocate(a0_d(n))
  call assign<<<1,n>>>(a0_d, 3.0)
  a = a0_d
  deallocate(a0_d)
  print *, "Device 0: ", a(1)

  istat = cudaSetDevice(1)
  allocate(a1_d(n))
  call assign<<<1,n>>>(a1_d, 4.0)
  a = a1_d
  deallocate(a1_d)
  print *, "Device 1: ", a(1)
end program minimal
```