GPU TECHNOLOGY CONFERENCE

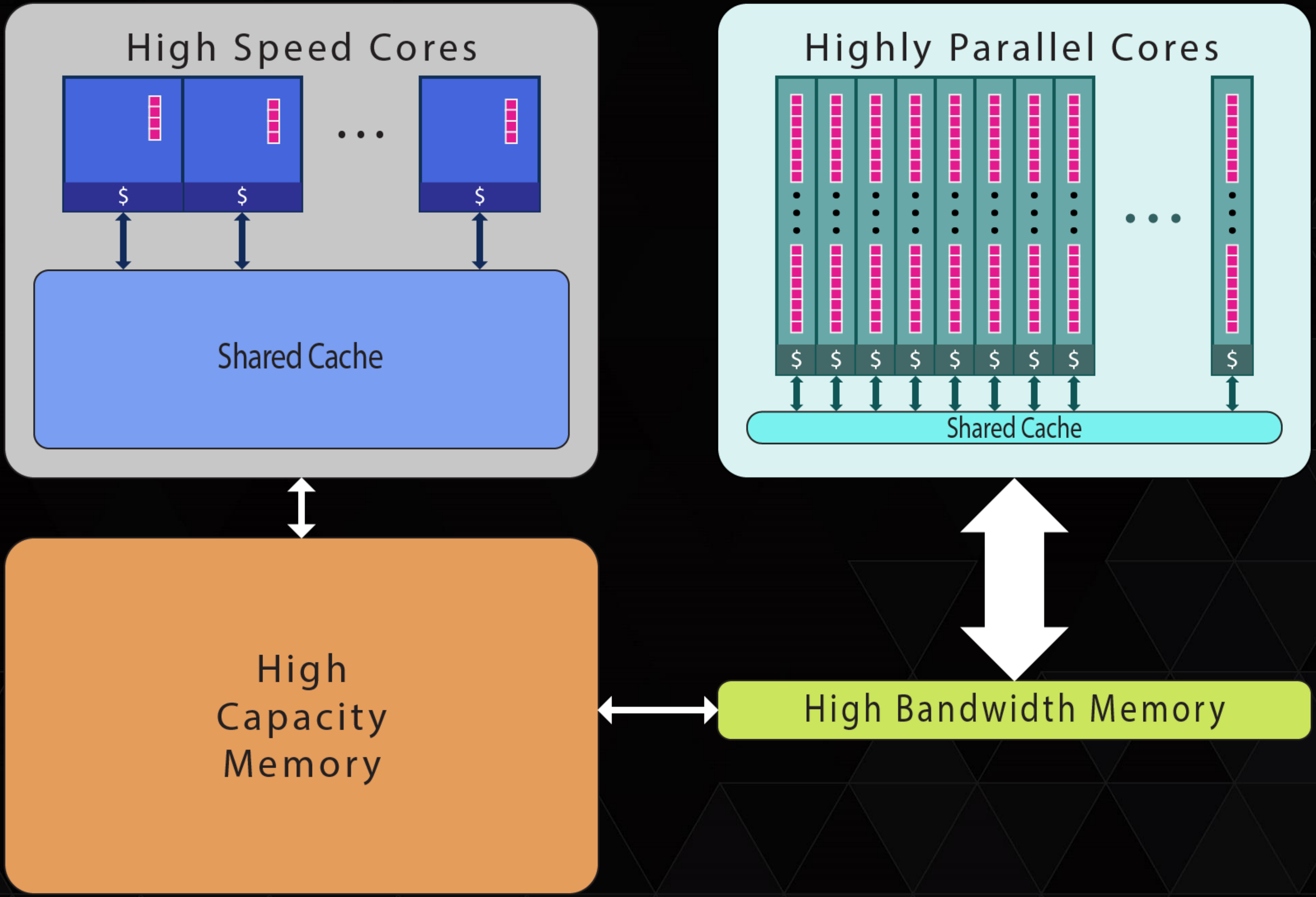
OpenACC for Fortran Programmers

Michael Wolfe
PGI compiler engineer

michael.wolfe@pgroup.com

Outline

- GPU Architecture
- Low-level GPU Programming and CUDA
- OpenACC Introduction
- Using the PGI Compilers
- Advanced Topics
  - Multiple Devices
  - Global Data
  - Procedures
  - Derived Types
  - Managed Memory
  - CUDA Fortran Interfacing



High Speed Cores
$
$
$
Shared Cache
Highly Parallel Cores
$
$
$
$
$
$
$
$
$
Shared Cache
High Capacity Memory
High Bandwidth Memory

CPU / Accelerator Differences

- Faster clock (2.5-3.5 GHz)
- More work per clock
  - Pipelining (deep)
  - Multiscalar (3-5)
  - SIMD width (4-16)
  - More cores (6-12)
- Fewer stalls
  - Large cache memories
  - Complex branch prediction
  - Out-of-order execution
  - Multithreading (2-4)

- Slower clock (0.8-1.0 GHz)
- More work per clock
  - Pipelining (shallow)
  - Multiscalar (1-2)
  - SIMD width (16-64)
  - More cores (15-60)
- Fewer stalls
  - Small cache memories
  - Little branch prediction
  - In-order execution
  - Multithreading (15-32)

Simple Fortran Example

```
real, allocatable :: a(:), b(:)
...
allocate(a(n),b(n))
...
call process( a, b, n )
...

subroutine process( a, b, n )
  real :: a(:), b(:)
  integer :: n, i
  do i = 1, n
    b(i) = exp(sin(a(i)))
  enddo
end subroutine
```

Low-Level Programming: CUDA Fortran

- Data Management
- Parallel Kernel Execution

```
real, allocatable :: a(:), b(:)
real, device, allocatable :: da(:),db(:)
...
allocate(a(n),b(n))
...
allocate(da(n),db(n))
da = a
nthrd = 128
nblk = (n+nthrd-1)/nthrd
call gprocess<<<nblk,nthrd>>>(da, db, n)
b = db
deallocate(da,db)
...
```

Low-Level Programming: CUDA Fortran

```
attributes(global) subroutine gprocess( a, b, n )
   real :: a(*), b(*)
   integer, value :: n
   integer :: i
   i = (blockidx%x-1)*blockdim%x + threadidx%x
   if( i <= n )
     b(i) = exp(sin(a(i)))
end subroutine
```

What is OpenACC?

- A set of directive-based extensions to C, C++ and Fortran that allow you to annotate regions of code and data for offloading from a CPU host to an attached Accelerator
- maintainable, portable, scalable

http://www.pgroup.com/lit/videos/pgi_openacc_webinar_july2012.html

http://www.pgroup.com/lit/videos/ieee_openacc_webinar_june2013.html

Higher-Level Programming: OpenACC

```
real, allocatable :: a(:), b(:)
...
allocate(a(n),b(n))
...
!$acc data copy(a,b)
call process( a, b, n )
!$acc end data
...
subroutine process( a, b, n )
  real :: a(:), b(:)
  integer :: n, i
  !$acc parallel loop
  do i = 1, n
    b(i) = exp(sin(a(i)))
  enddo
end subroutine
```

Data directives

- Data construct
  - allocates device memory
  - moves data in/out
- Update self(b)
  - copies device->host
  - aka update host(b)
- Update device(b)
  - copies host->device

```
real, allocatable :: a(:), b(:)
...
allocate(a(n),b(n))
...
!$acc data copyin(a) copyout(b)
...
 call process( a, b, n )
...
!$acc update self(b)
 call updatehalo(b)
!$acc update device(b)
...
!$acc end data
...
```

GPU TECHNOLOGY CONFERENCE

Data directives

- Enter data
  - like entry to data construct
  - allocates memory
  - moves data in
- Exit data
  - like exit from data construct
  - moves data out
  - deallocates memory

```
real, allocatable :: a(:), b(:)
...
allocate(a(n),b(n))
...
!$acc enter data copyin(a) create(b)
...
 call process( a, b, n )
...
!$acc update self(b)
 call updatehalo(b)
!$acc update device(b)
...
!$acc exit data delete(a) copyout(b)
...
```

Compute regions

- Parallel region
  - launches a device kernel
  - gangs / workers / vectors

```
subroutine process( a, b, n )
  real :: a(:), b(:)
  integer :: n, i
  !$acc parallel loop present(a,b)
  do i = 1, n
    b(i) = exp(sin(a(i)))
  enddo
end subroutine
```

Compute regions

- Parallel region
  - launches a device kernel
  - gangs / workers / vectors

```
subroutine process( a, b, n )
  real :: a(:,:), b(:,:)
  integer :: n, i, j
  !$acc parallel loop present(a,b)
  do j = 1, n
    !$acc loop vector
    do i = 1, n
      b(i,j) = exp(sin(a(i,j)))
    enddo
  enddo
end subroutine
```

Compute regions

- Kernels region
  - launches one or more device kernels
  - gangs / workers / vectors
  - more autoparallelization

```
subroutine process( a, b, n )
  real :: a(:,:), b(:,:)
  integer :: n, i, j
  !$acc kernels loop gang present(a,b)
  do j = 1, n
    !$acc loop vector
    do i = 1, n
      b(i,j) = exp(sin(a(i,j)))
    enddo
  enddo
end subroutine
```

Reductions

- reduction(operator:scalar)

  +, *, min, max
  iand, ior, ieor,
  .and., .or., .eqv., .neqv.

```
subroutine process( a, b, total, n )
  real :: a(:,:), b(:), total
  integer :: n, i, j
  real :: partial
  total = 0
  !$acc kernels loop gang present(a,b) &
              reduction(+:total)
  do j = 1, n
    partial = 0
    !$acc loop vector reduction(+:partial)
    do i = 1, n
      partial = partial + a(i,j)
    enddo
    b(i) = partial
    total = total + partial
  enddo
end subroutine
```

Collapse

- collapse(2)

```
subroutine process( a, b, total, n )
  real :: a(:,:), b(:,:), total
  integer :: n, i, j
  total = 0
  !$acc parallel loop collapse(2) &
      gang present(a,b) reduction(+:total)
  do j = 1, n
    do i = 1, n
      total = total + a(i,j)*b(i,j)
    enddo
  enddo
end subroutine
```

Independent / Auto

- parallel construct
  - independent
- kernels construct
  - auto

```
subroutine process( a, b, indx, n )
  real :: a(:,:), b(:)
  integer :: n, indx(:), i, j
  !$acc kernels loop present(a,b)
  do j = 1, n
    !$acc loop vector independent
    do i = 1, n
      a(indx(i),j) = b(i,j)*2.0
    enddo
  enddo
end subroutine
```

Private

- private to the gang / worker / vector lane executing that thread

```
subroutine process( a, b, indx, n )
  real :: a(:,:), b(:)
  integer :: n, indx(:), i, j, jt
  !$acc parallel loop present(a,b) &
   gang private(jt) independent
  do j = 1, n
    jt = indx(j)
    !$acc loop vector
    do i = 1, n
      a(i,jt) = b(i,j)*2.0
    enddo
  enddo
end subroutine
```

Atomic

- atomic update
- atomic read
- atomic write
- atomic capture

```
subroutine process( a, b, indx, n )
  real :: a(:,:), b(:)
  integer :: n, indx(:), i, j
  !$acc parallel loop present(a,b)
  do j = 1, n
    !$acc loop vector
    do i = 1, n
      !$acc atomic update
        b(indx(i)) = b(indx(i)) + a(i,j)
      !$acc end atomic
    enddo
  enddo
end subroutine
```

Update

- copy values between host and device copies

```
subroutine process( a, b, indx, n )
  real :: a(:), b(:)
  integer :: n, indx(:), i, j, jt
  !$acc data present(a,b)
  !$acc parallel loop
  do j = 1, n
    a(j) = b(j)*2.0
  enddo
  !$acc update self(a)
  !$acc end data
end subroutine
```

Using the PGI compilers

- pgfortran
- -acc
  - default -ta=tesla,host
- -ta=tesla[:suboptions...]
  - implies -acc
- -ta=radeon[:suboptions...]
  - implies -acc
- -ta=host
- -Minfo=accel

```
% pgfortran -ta=tesla a.f90 -Minfo=accel
% ./a.out

% pgfortran -acc -c b.f90 -Minfo=accel
% pgfortran -acc -c c.f90 -Minfo=accel
% pgfortran -acc -o c.exe b.o c.o
% ./c.exe
```

GPU TECHNOLOGY CONFERENCE

tesla suboptions

`-ta=tesla`	default: compiles for Fermi + Kepler + K20
`-ta=tesla:cc35`	compile for Kepler K20 only
`-ta=tesla:[no]rdc`	enable(default)/disable relocatable device code
`-ta=tesla:[no]fma`	enable/disable fused multiply-add
`-ta=tesla:cuda6.0|cuda6.5`	select toolkit version (6.0 default with PGI 15.1)
`-ta=tesla:O0`	override opt level: `O0,O1,O2,O3`
`-ta=tesla:keepgpu`	keeps `file.n001.gpu` generated file
`-ta=tesla -help`	print command line help

-Minfo=accel

```
% pgfortran -c -acc -Minfo=accel
process:
      4, Accelerator kernel generated
          5, !$acc loop gang ! blockidx%x
          7, !$acc loop vector(256) ! threadidx%x
      4, Generating copyout(b(:n,:n))
         Generating copyin(a(:n,:n))
         Generating Tesla code
      7, Loop is parallelizable
```

PGI_ACC_NOTIFY

```
% setenv PGI_ACC_NOTIFY 3
% a.out
upload CUDA data  file=/home/mwolfe/test2/15.03.test/a.f90
function=process line=6 device=0 variable=descriptor bytes=96
upload CUDA data  file=/home/mwolfe/test2/15.03.test/a.f90
function=process line=6 device=0 variable=descriptor bytes=96
upload CUDA data  file=/home/mwolfe/test2/15.03.test/a.f90
function=process line=6 device=0 variable=a bytes=10000
launch CUDA kernel  file=/home/mwolfe/test2/15.03.test/a.f90
function=process line=6 device=0 num_gangs=50 num_workers=1
vector_length=256 grid=50 block=256
download CUDA data  file=/home/mwolfe/test2/15.03.test/a.f90
function=process line=13 device=0 variable=b bytes=10000
```

PGI_ACC_TIME

```
% setenv PGI_ACC_TIME 1
% a.out
Accelerator Kernel Timing data
/home/mwolfe/test2/15.03.test/a.f90
  process  NVIDIA  devicenum=0
    time(us): 53
    6: data region reached 1 time
        6: data copyin transfers: 3
             device time(us): total=32 max=22 min=5 avg=10
        13: data copyout transfers: 1
             device time(us): total=15 max=15 min=15 avg=15
    6: compute region reached 1 time
        6: kernel launched 1 time
            grid: [50]  block: [256]
             device time(us): total=6 max=6 min=6 avg=6
            elapsed time(us): total=322 max=322 min=322 avg=322
```

GPU TECHNOLOGY CONFERENCE

Advanced: host_data

- replaces address of ‘a’ by device address of ‘a’
- mostly used in calls

```
!$acc data create( a(:,:) )
...
!$acc host_data use_device(a)
  call MPI_Send( a, n*n, ... )
!$acc end host_data
```

Advanced: Multiple Threads

- Nest OpenACC within OpenMP regions
- All threads share context on the device
- Race conditions!
- no omp and acc on same loop

```
!$omp parallel
 ...
!$acc data copyin(a(:,:), b(:,:))
 ...
  !$omp parallel do
  do i = 1, n
    !$acc parallel loop
    do j = 1, n
      a(i,j) = sin(b(i,j))
    enddo
  enddo
 ...
!$acc end data
```

Advanced: Multiple Devices

- acc_set_device_num()
- MPI Ranks attach to different device
- OpenMP threads attach to different device
- Single thread switches between devices

```
call MPI_Comm_Rank( MPI_COMM_WORLD, rank )
ndev = acc_get_num_devices(acc_device_nvidia)
idev = mod(rank,ndev)
call acc_set_device_num(idev,acc_device_nvidia)
...
!$acc data copy(a)
  ...
```

Advanced: Declare global data

- Global data
- Subprogram scope data

```
module mymod
  real :: coef
  !$acc declare create(coef)
  real, allocatable :: value(:)
  !$acc declare create(value)
end module
subroutine s
  use mymod
  !$acc parallel loop
  do i = 1, n
    value(i) = coef*value(i)
  enddo
end subroutine
```

Advanced: Procedures

- Compile subprograms for device execution
- Specify whether the subprogram has parallel loops
- routine seq implies no parallelism
- within a single file, nordc works
- across files, must use rdc (default)

```
module mymod
  real :: coef
  !$acc declare create(coef)
  real, allocatable :: value(:)
  !$acc declare create(value)
contains
  subroutine initvalue( ri, rs )
    !$acc routine gang
    real :: ri, rs
    integer :: i
    !$acc loop gang vector
    do i = 1, ubound(value,1)
      value(i) = ri + (i-1)*rs
    enddo
  end subroutine
end module
```

Asynchronous Operations

- async(q) clause
  - enter/exit data
  - update
  - parallel/kernels
- wait directive
  - waits for all async queues
- wait(q) directive
  - waits only for queue q
- wait(q) async(r) together

```
!$acc enter data copyin(a) async(1)
!$acc enter data copyin(b) async(1)
!$acc parallel loop async(1)
   do i = 1, n
     a(i) = a(i) + 1
   enddo
!$acc update self async(1)
...
!$acc wait(1)
  s = sum(a)
```

Asynchronous Operations

- async(q) clause
  - enter/exit data
  - update
  - parallel/kernels
- wait directive
  - waits for all async queues
- wait(q) directive
  - waits only for queue q
- wait(q) async(r) together
  - won't start until queue q is ready
  - host program continues

```
!$acc enter data copyin(a) async(1)
!$acc enter data copyin(b) async(2)
!$acc parallel loop wait(2) async(1)
   do i = 1, n
     a(i) = a(i) + 1
   enddo
!$acc update self async(1)
...
!$acc wait(1)
  s = sum(a)
```

Advanced: Derived Types

- Arrays of derived type just work
- Derived type with allocatable array members require some work
- Array of derived type with allocatable array members require more work

```
module gdatamod
  type point
    real :: x, y, z
  end type
  type gdata
    type(point), allocatable :: &
                   loc(:), vel(:)
    real, allocatable :: weight(:)
  end type
end module
  use gdatamod
  type(gdata) :: d
  allocate( d%points(n), g%weights(n) )
  do i = 1, n
    d%loc(i)%x = d%loc(i)%x + d%vel(i)%x
    ...
  enddo
```

Advanced: Derived Types

```
module gdatamod
  type point
    real :: x, y, z
  end type
  type gdata
    type(point), allocatable :: &
        loc(:), vel(:)
    real, allocatable :: &
        weight(:)
  end type
end module
```

```
type(gdata) :: d
...
!$acc enter data copyin(d)
!$acc enter data copyin(d%loc)
!$acc enter data copyin(d%vel)

!$acc parallel loop present(d)
do i = 1, n
  d%loc(i)%x = d%loc(i)%x &
             + d%vel(i)%x
  ...
enddo

!$acc update self(d%loc)
```

Advanced: Managed Memory

- CUDA Unified (managed) Memory - 64-bit Linux only
- One address space for host and device
- Data allocated in managed memory
  - is moved to GPU when a kernel is launched
  - is moved back to system memory at host page fault
- Limited to device memory size
- `-ta=tesla:managed`

```
module mymod
  real :: coef
  !$acc declare create(coef)
  real, allocatable :: value(:)
contains
  subroutine initvalue( ri, rs )
    !$acc routine gang
    real :: ri, rs
    integer :: i
    !$acc loop gang vector
    do i = 1, ubound(value,1)
      value(i) = ri + (i-1)*rs
    enddo
  end subroutine
end module
```

Advanced: Interoperability

- CUDA data in OpenACC compute constructs
- OpenACC data in CUDA kernel launches
- OpenACC calling CUDA device routines

```
module mymod
  real, allocatable, device :: value(:)
contains
  subroutine initvalue( ri, rs )
    real :: ri, rs
    integer :: i
    !$acc parallel loop
    do i = 1, ubound(value,1)
      value(i) = ri + (i-1)*rs
    enddo
  end subroutine
end module
```

Advanced: Interoperability

- CUDA data in OpenACC compute constructs
- OpenACC data in CUDA kernel launches
- OpenACC calling CUDA device routines

```
module mymod
  real, allocatable :: value(:)
contains
  attributes(global) &
  subroutine ss(a)
    real :: a(*)
    ...
  end subroutine
  subroutine initvalue( value, n )
    real :: value(*)
    !$acc data present(value)
    call ss<<<n/64,64>>>( value )
    !$acc end data
  end subroutine
end module
```

Advanced: Interoperability

- CUDA data in OpenACC compute constructs
- OpenACC data in CUDA kernel launches
- OpenACC calling CUDA device routines

```
module mymod
  real, allocatable, device :: value(:)
contains
  attributes(device) real function ss(a,j)
    real :: a(*)
    integer, value :: j
    ...
 end function
 subroutine initvalue( value, ini, n )
    real :: value(*), ini(*)
    integer :: i, n
    !$acc parallel loop present(value,ini)
    do i = 1, ubound(value,1)
      value(i) = ss( ini, i )
    enddo
 end subroutine
end module
```

Common Errors or Problems

- Access array out of bounds
- Data not present
- Stale Data
- Async error
- Roundoff error changes
- Parallelization errors
- Bad data clause limits
- Missing data clause
- Missing update directive
- Missing wait, >1 async queue
- Reduction, FMA
- bad parallel or loop independent

Summary

- Data: data construct, enter data / exit data, update device / update self
- Compute: parallel/kernels, loop directive, reduction clause, atomic
- Global data and Procedures: acc declare and acc routine
- Asynchronous operations: async clause, wait directive, wait clause
- Interoperability: data sharing, CUDA device routines
- Future: unified memory (even better than managed), deep copy
- More information:
  - www.openacc.org
  - www.pgroup.com/openacc
  - michael.wolfe@pgroup.com