Introduction to OpenACC

GTC 2012

Brent Leback
Brent.Leback@pgroup.com
http://www.pgroup.com
What is OpenACC?

- PGI Introduced its Accelerator Programming Model in 2008
  - It has gone through a few iterations, is currently at version 1.3
- Goal then and now is to produce a higher-level model than CUDA, aimed at scientists and engineers
- Wanted something that is directive-based, in the spirit of OpenMP
- A few alternate but similar models were introduced later
  - CAPS, Cray
- OpenMP committee attempted to standardize a model in 2010
- Meeting of the minds last fall, OpenACC 1.0 announced at SC11
- See www.openacc.org
GPU Architecture Features

- Optimized for high degree of regular parallelism
- Outer do-all parallelism is fully parallel across the multiprocessors
- SIMD parallelism within a multiprocessor, which can synchronize and share data
- High bandwidth memory, support for ECC
- Highly multithreaded (slack parallelism) with hardware thread scheduling
- Non-coherent hw data caches, sw managed shared memory
- No multiprocessor memory model guarantees
  - Low-level atomic functions available, but not generally recommended
GPU Programming Constants

The Program must:

- Initialize/Select the GPU to run on
- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
- Gather results from GPU
- Deallocate data
CUDA Fortran, an Explicit Language

use vaddmod
real, device, dimension(:), allocatable :: da, db, dc

istat = cudaSetDevice(0)
allocate( da(1:n), db(1:n), dc(1:n) )

db = b
dc = c

call vadd<<<min((n+255)/256,65535),256>>>( da, db, dc, n )

a = da
deallocate( da, db, dc )
Implicit Model

```c
!$acc kernels loop
do  i = 1, n
   a(i) = b(i) + c(i)
enddo

#pragma acc kernels loop
for( i = 0; i < n; ++i )
a[i] = b[i] + c[i];
```

Compiler determines

- Need to allocate a, b, c of length n on the device
- Copyin of b and c
- Need to generate and call a kernel for the specified operation, decide on a launch configuration
- Copyout a
- Deallocate a, b, and c
 DATA DIRECTIVES

C    INITIALIZE CONSTANTS AND ARRAYS

C

CALL ALLOC

!$ACC DATA CREATE(U(:NP1,:MP1), V(:NP1,:MP1))
!$ACC& CREATE(UNEW(:NP1,:MP1), VNEW(:NP1,:MP1))
!$ACC& CREATE(PNEW(:NP1,:MP1), UOLD(:NP1,:MP1))
!$ACC& CREATE(VOLD(:NP1,:MP1), POLD(:NP1,:MP1))
!$ACC& CREATE(CU(:NP1,:MP1), CV(:NP1,:MP1))
!$ACC& CREATE(P(:NP1,:MP1), Z(:NP1,:MP1))
!$ACC& CREATE(H(:NP1,:MP1), PSI(:NP1,:MP1))

CALL INITIAL

- Defines a region where arrays should be allocated on the device
- Often just one large data region per program
- Clauses define copy behavior
- Use present clauses in subprograms
#pragma acc kernels loop
  copyin(b[0:n*m]) copy(a[0:n*m])
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i*m+j] = w0 * b[i*m+j] +
    w1*(b[(i-1)*m+j] + b[(i+1)*m+j] + b[i*m+j-1] + b[i*m+j+1]) +
    w2*(b[(i-1)*m+j-1] + b[(i-1)*m+j+1] + b[(i+1)*m+j-1] + b[(i+1)*m+j+1]);

- Most like the PGI Accelerator region
- Contains loop constructs, can generate multiple kernels
- Compiler is free to schedule kernels onto hardware, but schedule can be directed with clauses
- Currently requires tightly nested loops, no undersubscribed gangs
The Parallel Construct

```c
!$acc parallel
! Do some redundant gang work here
!$acc loop gang
do j = 1, n
  p1 = posin(j,1)
p2 = posin(j,2)
p3 = posin(j,3)
f1 = 0.0; f2 = 0.0; f3 = 0.0
!$acc loop worker, reduction(+:f0,f1,f2)
do i = 1, n
  r1 = posin(i,1) - p1
  r2 = posin(i,2) - p2
  r3 = posin(i,3) - p3
  distsqr = r1*r1 + r2*r2 + r3*r3
  . . .
```

- Most like an OpenMP Parallel region
- Contains loop constructs, generates one kernel
- Compiler schedule is fixed by `num_gangs` and `num_workers`
- One worker in each gang executes redundantly until a work-sharing loop is encountered
Parallel vs. Kernels

#pragma acc kernels
{
    for( i = 1; i < n-1; ++i )
        x[i] = 0.5*y[i] + 0.25*(y[i-1] + y[i+1]);
    for( i = 1; i < n-1; ++i )
        y[i] = 0.5*x[i] + 0.25*(x[i-1] + x[i+1]);
}
#pragma acc parallel
{
    #pragma acc loop
    for( i = 1; i < n-1; ++i )
        x[i] = 0.5*y[i] + 0.25*(y[i-1] + y[i+1]);
    #pragma acc loop
    for( i = 1; i < n-1; ++i )
        y[i] = 0.5*x[i] + 0.25*(x[i-1] + x[i+1]);
}
```c
void domany(...){
    saxpy( n, a, x, y );
}

void saxpy( int n, float a,
            float* x, float* restrict y ){
    int i;

    #pragma acc kernels loop \
        copyin(x[0:n]) copy(y[0:n])
    for( i = 1; i < n; ++i )
        y[i] += a*x[i];
}
```
No Data Region - Fortran

subroutine domany(...)
   call saxpy( n, a, x, y )
end subroutine

subroutine saxpy( n, a, x, y )
   integer :: n
   real :: a, x(*), y(*)
   integer :: i
   !$acc kernels loop &
      copyin(x(1:n)) copy(y(1:n))
   do i = 1, n
      y(i) = y(i) + a*x(i)
   enddo
end subroutine
Data Region across Procedures

```c
void domany(...){

#pragma acc data \
   copy(x[0:n],y[0:n])
{
   saxpy( n, a, x, y );
}

void saxpy( int n, float a,
           float* x, float* restrict y ){
    int i;

#pragma acc kernels loop \
    present(x[0:n], y[0:n])
    for( i = 1; i < n; ++i )
    y[i] += a*x[i];
}
```
subroutine domany(...)!

 !$acc data copy( x(:), y(:) )
 call saxpy( n, a, x, y )
 !$acc end data

 subroutine saxpy( n, a, x, y )
 integer :: n
 real :: a, x(*), y(*)
 integer :: i

 !$acc kernels loop &
 present(x(1:n),y(1:n))
 do i = 1, n
 y(i) = y(i) + a*x(i)
 enddo
 end subroutine
void domany(...){

#pragma acc data \
  copy(x[0:n],y[0:n])
{
  saxpy( n, a, x, y );
}

saxpy( n, a, x2, y2 );
}

void saxpy( int n, float a, 
  float* x, float* restrict y ){
  int i;

#pragma acc kernels loop \
  present_or_copyin(x[0:n])\n  present_or_copy(y[0:n])
  for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
Data Region across Procedures

subroutine domany(...)  subroutine saxpy( n, a, x, y )
  !$acc data copy( x(:), y(:) )
  call saxpy( n, a, x, y )
  !$acc end data
  !$acc kernels loop &
  present_or_copyin(x(1:n)) &
  present_or_copyin(y(1:n))
  do i = 1, n
    y(i) = y(i) + a*x(i)
  enddo
end subroutine
subroutine domany(...)  subroutine saxpyy( n, a, x, y )

!$acc data copy( x(:), y(:) )
    integer :: n
    real :: a, x(*), y(*)
    integer :: i

    call saxpy( n, a, x, y )
    !$acc kernels
    do i = 1, n
    y(i) = y(i) + a*x(i)
    enddo
end subroutine

!$acc end data

call saxpy( n, a, x2, y2 )
end subroutine
subroutine domany(...)  subroutine saxpy( n, a, x, y )  

!$acc data copy( x(:,:), y(:) )  integer :: n
  do j = 1, m
    real :: a, x(:), y(:)
    integer :: i
    !$acc kernels loop
    call saxpy(n, a, x(:,j), y)
    do i = 1, n
      !$acc kernels loop
      y(i) = y(i) + a*x(i)
    enddo
  enddo
end subroutine

do j = 1, m
  call saxpy(n, a, x(:,j), y)
enddo
!$acc end data

!$acc end data
Update

Original Code:

```c
for( timestep=0;...){
  ...compute...

  MPI_SENDRECV( x, ... )

  ...adjust...
}
```

```c
#pragma acc data \
  copy(x[0:n])...
{
  for( timestep=0;...){
    ...compute on device...
    #pragma update host \ 
    (x[0:n])
    MPI_SENDRECV( x, ... )
    #pragma update device \ 
    (x[0:n])
    ...adjust on device
    ...
  }
}
```
Update

- Update directive assumes present
- You can specify subarrays
- Non-contiguous data may be slower
- You may want to add code to move data

```c
#pragma acc data \
  copy(x[0:n])...
{
  for( timestep=0;...){
    ...compute on device...
    #pragma update host \
    (x[0:n])
    MPI_SENDRECV( x, ... )
    #pragma update device \
    (x[0:n])
    ...adjust on device
  }
}
```
Asynchronous Clauses

- synchronous – directive / construct does not complete until action is complete
- asynchronous – program will continue beyond directive / construct before action is complete
- Use the wait directive to synchronize
void domany(...){

#pragma acc data \
    create(x[0:n],y[0:n])
{
    #pragma acc update device \
        (x[0:n], y[0:n]) async
        saxpy( n, a, x, y );
    #pragma acc update host \
        (y[0:n]) async
        ....
    #pragma acc wait
}

void saxpy( int n, float a, float* x, float* restrict y ){
    int i;

    #pragma acc kernels loop async
    for( i = 1; i < n; ++i )
        y[i] += a*x[i];
}
subroutine domany(...)  subroutine saxpy( n, a, x, y, j )
  !$acc data copy( y(:,,:), x(:) )
  !$acc kernels loop async(j)
  do j = 1, m
    call saxpy(n,a,x,y(:,j),j)
    !$acc wait ! waits for all
  !$acc end data
  do i = 1, n
    y(i) = y(i) + a*x(i)
  enddo
end subroutine
subroutine smoothiter( a, b, w, n, m, js, je, usegpu )
  real, dimension(:,,:) :: a, b
  real, intent(in) :: w
  integer, intent(in) :: n, m, js, je
  logical, intent(in) :: usegpu

  !$acc kernels loop present(a(:,js-1:je+1),b(:,js-1:js+1)) &
  async if(usegpu)
  do j = js, je
    do i = 2, n-1
      a(i,j) = b(i,j) + &
        w * (b(i-1,j) + b(i+1,j) + b(i,j-1) + b(i,j+1))
    enddo
  enddo
end subroutine
js = (m*pct)/100
!$acc data copy( a(:,1:js+1), b(:,1:js+1) )
do iter = 1, maxiters
    call smoothiter( a, b, w, n, m, 2, js, .true. )
call smoothiter( a, b, w, n, m, js+1, m-2, .false. )
!$acc update host( a(:,js) ) device( a(:,js+1) ) async
!$acc wait
call smoothiter( b, a, w, n, m, 2, js, .true. )
call smoothiter( b, a, w, n, m, js+1, m-2, .false. )
!$acc update host( b(:,js) ) device( b(:,js+1) ) async
!$acc wait
enddo
!$acc end data
Data Caching

!$acc kernels loop present(a(:,js-1:je+1),b(:,js-1:js+1))
  do j = js, je
    do i = 2, n-1
      !$acc cache( b(i-1:i+1,j-1:j+1) )
      a(i,j) = b(i,j) + &
      w * (b(i-1,j) + b(i+1,j) + b(i,j-1) + b(i,j+1))
    enddo
  enddo
end subroutine
Multiple GPUs – Use MPI or OpenMP

```c
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(2)
{
    int i = omp_get_threadnum();
    acc_set_device_num( i, acc_device_nvidia );
    #pragma acc data copy...
    {
    }
}
```
Multiple GPUs – Use MPI or OpenMP

```c
#include <openacc.h>
#include <mpi.h>

int myrank;
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
int numdev = acc_get_num_devices( acc_device_nvidia );
int i = myrank % numdev;
acc_set_device_num( i, acc_device_nvidia );
```
Procedure calls on the device

```c
#pragma acc parallel \
    copy(x[0:n],y[0:n])
{
    saxpy( n, a, x, y );
}
```
Procedure calls on the device

```c
#pragma acc parallel \
    copy(x[0:n], y[0:n])
{
    saxpy( n, a, x, y );
}

void saxpy( int n, float a, float* x, float* restrict y ){
    int i;

    #pragma acc loop
    for( i = 1; i < n; ++i )
        y[i] += a*x[i];
}
```

- Use inlining
- PGI: `-Minline[=levels:3]`
## PGI C Intrinsics

**PGI C:** `#include <accelmath.h>`

<table>
<thead>
<tr>
<th>acos</th>
<th>asin</th>
<th>atan</th>
<th>atan2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos</td>
<td>cosh</td>
<td>exp</td>
<td>fabs</td>
</tr>
<tr>
<td>fmax</td>
<td>fmin</td>
<td>log</td>
<td>log10</td>
</tr>
<tr>
<td>pow</td>
<td>sin</td>
<td>sinh</td>
<td>sqrt</td>
</tr>
<tr>
<td>tan</td>
<td>tanh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acosf</td>
<td>asinf</td>
<td>atanf</td>
<td>atan2f</td>
</tr>
<tr>
<td>cosf</td>
<td>coshf</td>
<td>expf</td>
<td>fabsf</td>
</tr>
<tr>
<td>fmaxf</td>
<td>fminf</td>
<td>logf</td>
<td>log10f</td>
</tr>
<tr>
<td>powf</td>
<td>sinf</td>
<td>sinhf</td>
<td>sqrtf</td>
</tr>
<tr>
<td>tanf</td>
<td>tanhf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## PGI Fortran Intrinsics

<table>
<thead>
<tr>
<th>abs</th>
<th>acos</th>
<th>aint</th>
<th>asin</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan</td>
<td>atan2</td>
<td>cos</td>
<td>cosh</td>
</tr>
<tr>
<td>dble</td>
<td>exp</td>
<td>iand</td>
<td>ieor</td>
</tr>
<tr>
<td>int</td>
<td>ior</td>
<td>log</td>
<td>log10</td>
</tr>
<tr>
<td>max</td>
<td>min</td>
<td>mod</td>
<td>not</td>
</tr>
<tr>
<td>real</td>
<td>sign</td>
<td>sin</td>
<td>sinh</td>
</tr>
<tr>
<td>sqrt</td>
<td>tan</td>
<td>tanh</td>
<td></td>
</tr>
</tbody>
</table>
other functions

- PGI libm routines
  - use libm
  - #include <accelmath.h>

- PGI device builtin routines
  - use cudadevice
  - #include <cudadevice.h>
Mixing OpenACC with CUDA C

```c
#pragma acc data copy( x[0:n] )
...
#pragma acc host_data use_device(x)
{
    uses_cuda_pointer( x );
}
...
}```
Mixing OpenACC with CUDA C

cudaMalloc( &x, sizeof(float)*n );
...
#pragma acc data deviceptr(x, y)
{
    for( i = 0; i < n; ++i )
        y[i] += a * x[i];
}
Mixing OpenACC with CUDA Fortran (PGI)

module mymod
    contains
    subroutine usesdev( x )
        real, dimension(:,), device :: x
        ...
    end subroutine
end module
...
use mymod
!$acc data copy( y(:) )
...
call usesdev( y )
...
!$acc end data
Mixing OpenACC with CUDA Fortran (PGI)

module mymod
  real, dimension(:), allocatable, device :: x
end module
...
use mymod
!$acc data copy( y(:) ) ! no need for 'x' here
...
!$acc kernels loop
  do i = 1, n
    y(i) = y(i) + a*x(i)
  enddo
...
!$acc end data
module mymod
  real, dimension(:,), allocatable, device :: x
contains
  attributes(device) subroutine devproc(...)
...  
  end subroutine
subroutine hostproc(...)  
    !$acc parallel  
    do i = 1, n  
      call devproc(a(i))  
    enddo  
    !$acc end parallel  
  end subroutine
end module
C-specific Features and Issues

- Precision matters
  - `-Mfcon` flag (PGI)

- Pointer disambiguation matters

```c
float* restrict a;
```
PGI-Specific Features and Issues

- new functions
- 2D C arrays
- compiler feedback
- async on data construct
- CUDA Fortran integration
- compiler suboptions
- PGI Unified Binary
for( ptr = head; ptr;  ptr = ptr->next )
    acc_copyin( ptr->y, sizeof(float)*ptr->size );
...
#pragma acc data copyin( x[0:n] )
{
    for( ptr = head; ptr; ptr = ptr->next )
        saxpy( n, a, x, ptr->y );
}

for( ptr = head; ptr;  ptr = ptr->next )
    acc_copyout( ptr->y, sizeof(float)*ptr->size );
Async on Data construct

```c
void domany(...){
    
    #pragma acc data async \
    copy(x[0:m][0:n],y[0:n])
    {
        for( j = 0; j < m; ++j )
            saxpy( n, a, x[j], y );
    }
    
    #pragma acc wait
}

void saxpy( int n, float a,
            float* x, float* restrict y ){
    int i;
    
    #pragma acc kernels loop async
    for( i = 1; i < n; ++i )
        y[i] += a*x[i];
}
```
CUDA Fortran integration

- data with device attribute can be used in OpenACC constructs
- data transfers with pinned attribute will be faster
- OpenACC parallel/kernels may call CUDA library
- OpenACC parallel/kernels may call user device subprograms
  - in same module
- OpenACC data may be passed to device arguments
Compiler Suboptions

-acc enables OpenACC recognition
-\texttt{ta=nvidia} sets target accelerator (default)
-\texttt{ta=nvidia,cc1x cc10 cc11 cc12 cc13 cc2x cc20 [cc3x cc30]}
  \begin{itemize}
  \item sets compute capability(ies)
  \end{itemize}
-\texttt{ta=nvidia,fastmath} uses fast math versions (less accurate)
-\texttt{ta=nvidia,cuda4.0 cuda4.1 [cuda4.2]} sets toolkit version
-\texttt{ta=nvidia,nofma} avoids use of fused mul-add (precision diffs)
-\texttt{ta=nvidia,O0 O1 O2 O3} sets device code opt level
-\texttt{ta=nvidia,keepgpu} lets you look at generated GPU code
Other Compiler/Runtime Tips

- The compiler flag to enable OpenACC is
  - -acc [ = strict | verystrict ]
  - Use this in combination with the –ta=nvidia target options
- Use the –Minfo=accel option to enable compiler feedback
- PGI Accelerator Model and OpenACC can coexist in the same program, as can CUDA Fortran and OpenACC. They can share features.
- Use the PGI_ACC_TIME environment variable to get a quick accounting of data transfer between host and device, and some quick kernel statistics
PGI Unified Binary

-tp=sandybridge,barcelona
  - two versions of relevant routines, one with AVX (for instance)

-ta=nvidia,host
  - two versions of relevant routines, one host only, one GPU accelerated
    acc_set_device_type( acc_device_nvidia )
    acc_init( acc_device_nvidia )
    acc_set_device_num( acc_device_nvidia, 0 )
    or acc_device_host
OpenACC Evolution, Implementations

- C++
- New targets: multicore, MIC, ATI, other...
- More tools support
- More interoperability with CUDA / OpenCL
- Separate compilation, linker, libraries
- Nested parallelism
- Multiple GPUs
New/Upcoming Features in PGI Accelerator Compilers

- CUDA-x86 Compilers officially released in January
- PGI OpenCL Compilers for ARM announced in March
- CUFFT interface modules for CUDA Fortran in PGI 12.5
- Support for OpenACC parallel construct in PGI 12.5
- Support for CUDA 4.2 in PGI 12.6
- CUDA Fortran support for textures coming soon
- PGI OpenACC release 1.0 in June or July 2012
- Work on generating PTX directly, with dwarf, is underway
- PGI Accelerator Model 2.0 specification will be out by ISC12
Where to get help

- OpenACC Forum – www.openacc.org/forum
- OpenACC documentation – www.openacc.org/downloads
- PGI Licensed Customer Support - trs@pgroup.com
- PGI Articles – www.pgroup.com/resources/articles.htm
  www.pgroup.com/resources/accel.htm
- CUDA Fortran Reference Guide –
Copyright Notice

© Contents copyright 2012, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.