Towards an integration of directive-based and explicit accelerator programming models

douglas.miles@pgroup.com

The Portland Group (PGI)

October 2012
Overview

- PGI Accelerator / OpenACC roadmap
- An abstract machine for OpenACC programmers
- Directives are necessary, but are they sufficient?
- What’s so great about CUDA Fortran?
- Elements of an integrated model
PGI Accelerator / OpenACC Roadmap*

Platforms

- CUDA 4.1
- CUDA 4.2
- CUDA 5.0
- CUDA 5.x
- CUDA-x86
- NVIDIA Kepler
- NVIDIA Kepler-2
- NVIDIA Maxwell
- NVIDIA Denver

Accelerator

- PGI Acc Profiling
- VS 2012 Integration
- PGI Acc on LLVM
- F90 Ptr Support
- Acc Region Func Calls
- Nested Acc Compute
- Multi-Device improvements
- C++ openACC
- Code Generator Tuning
- Device DWARF Gen
- CUDA Fortran Textures
- PGI Acc 2.0 Spec
- OpenACC 2nd HPC target
- CUDA Fortran Textures
- PGI Acc 2.0 for NVIDIA
- PGI Acc 2.0 2nd HPC target
- PGI Acc 2.0 Spec

Features

- PGI Accelerator 2nd HPC Platform
- CUDA Fortran Textures
- CUDA Fortran Textures
- CUDA Fortran Textures

*PGI roadmaps are subject to change without notice
OpenACC Abstract Machine Architecture
Directives are necessary, but are they sufficient?

- What about library developers, other experts?
- Accelerator performance cliff
- Predictability
- Compiler maturity
- Portability, OpenACC interoperability of explicit models
What’s so great about CUDA Fortran?
1. The host code – it looks normal!

```fortran
real, device, allocatable, dimension(:,:) ::
   Adev, Bdev, Cdev

allocate (Adev(N,M), Bdev(M,L), Cdev(N,L))
Adev = A(1:N,1:M)
Bdev = B(1:M,1:L)
call mm_kernel <<<dim3(N/16,M/16),dim3(16,16)>>>
   ( Adev, Bdev, Cdev, N, M, L)
C(1:N,1:L) = Cdev
deallocate ( Adev, Bdev, Cdev )
```

```fortran
attributes(global) subroutine mm_kernel
   ( A, B, C, N, M, L )
real :: A(N,M), B(M,L), C(N,L), Cij
integer, value :: N, M, L
integer :: i, j, kb, k, tx, ty
real, shared :: Asub(16,16), Bsub(16,16)
tx = threadidx%x
ty = threadidx%y
i = blockIdx%x * 16 + tx
j = blockIdx%y * 16 + ty
Cij = 0.0
do kb = 1, M, 16
   Asub(tx,ty) = A(i,kb+tx-1)
   Bsub(tx,ty) = B(kb+ty-1,j)
call syncthreads()
do k = 1,16
   Cij = Cij + Asub(tx,k) * Bsub(k,ty)
call syncthreads()
endo
ddo
C(i,j) = Cij
end subroutine mmul_kernel
```
2. OpenACC interoperability

module mymod
    real, dimension(:), allocatable, device :: xDev
end module
...
use mymod
...
allocate( xDev(n) )
call init_kernel <<<dim3(n/128),dim3(128)>>>(xDev, n)
...
!$acc data copy( y(:) ) ! no need to copy 'xDev'
...
!$acc kernels loop
  do i = 1, n
    y(i) = y(i) + a*xDev(i)
  enddo
...
!$acc end data
3. !$CUF kernel directives

module madd_device_module
use cudafor
contains
subroutine madd_dev(a,b,c,sum,n1,n2)
  real,dimension(:,:),device :: a,b,c
  real :: sum
  integer :: n1,n2
  type(dim3) :: grid, block
  !$cuf kernel do (2) <<<(*,*),(32,4)>>>
  do j = 1,n2
    do i = 1,n1
      a(i,j) = b(i,j) + c(i,j)
      sum = sum + a(i,j)
    enddo
  enddo
end subroutine
end module

module madd_device_module
use cudafor
implicit none
contains
attributes(global) subroutine madd_kernel(a,b,c,blocksum,n1,n2)
  real, dimension(:,:) :: a,b,c
  real, dimension(:) :: blocksum
  integer, value :: n1,n2
  integer :: i,j,tindex,tneighbor,bindex
  real :: mysum
  real, shared :: bsum(256)
  ! Do this thread's work
  mysum = 0.0
  do j = threadidx%y + (blockidx%y-1)*blockdim%y, n2, blockdim%y*griddim%y
    do i = threadidx%x + (blockidx%x-1)*blockdim%x, n1, blockdim%x*griddim%x
      a(i,j) = b(i,j) + c(i,j)
      mysum = mysum + a(i,j) ! accumulates partial sum per thread
    enddo
  enddo
  ! Now add up all partial sums for the whole thread block
  ! Compute this thread's linear index in the thread block
  ! We assume 256 threads in the thread block
  ! Store this thread's partial sum in the shared memory block
  bsum(tindex) = mysum
  call syncthreads()
  ! Accumulate all the partial sums for this thread block to a single value
  blocksum = blocksum + bsum(1)
  ! Store the partial sum for the thread block
  bsum(1) = blocksum
  call syncthreads()
end subroutine
end module

! Add up partial sums for all thread blocks to a single cumulative sum
attributes(global) subroutine madd_sum_KERNEL(blocksum,dsum,nb)
  real, dimension(:) :: blocksum
  real :: dsum
  integer, value :: nb
  real, shared :: bsum(256)
  integer :: tindex,tneighbor,i
  ! Again, we assume 256 threads in the thread block
  ! accumulate a partial sum for each thread
  tindex = threadidx%x
  bsum(tindex) = 0.0
  do i = tindex, nb, blockdim%x
    bsum(tindex) = bsum(tindex) + blocksum(i)
  enddo
  call syncthreads()
  ! This code is copied from the previous kernel
  ! Accumulate all the partial sums for this thread block to a single value
  ! Since there is only one thread block, this single value is the final result
  blocksum = blocksum + bsum(1)
end subroutine

subroutine madd_dev(a,b,c,sum,n1,n2)
  real,dimension(:,:),device :: a,b,c
  real, device :: dsum
  real, device :: blocksum
  integer :: n1,n2,nb
  type(dim3) :: grid, block
  integer :: r
  ! Compute grid/block size; block size must be 256 threads
  grid = dim3((n1+31)/32, (n2+7)/8, 1)
  block = dim3(32,8,1)
  nb = grid%x * grid%y
  allocate(blocksum(1:nb))
  call madd_kernel<<< grid, block >>>(a,b,c,blocksum,n1,n2)
  call madd_sum_KERNEL<<< 1, 256 >>>(blocksum,dsum,nb)
  r = cudaThreadSynchronize() ! don't deallocate too early
  deallocate(blocksum)
end subroutine
use cublas

real(4), device :: xd(N)
real(4) x(N)
call random_number(x)

! On the device
allocate(xd(N))
xd = x
j = isamax(N,xd,1)

! On the host, same name
k = isamax(N,x,1)

module cublas
  ! isamax
  interface isamax
    integer function isamax &
    (n, x, incx)
    integer :: n, incx
    real(4) :: x(*)
  end function
  
  integer function isamaxcu &
    (n, x, incx) bind(c, &
     name='cublasIsamax')
  integer, value :: n, incx
  real(4), device :: x(*)
end interface
end module cublas
5. Encapsulation

- Isolate device data and accelerator kernel declarations in Fortran modules

```fortran
module mm
  real, device, allocatable :: a(:)
  real, device :: x, y(10)
  real, constant :: c1, c2(10)
  integer, device :: n
contains
  attributes(global) subroutine s( b )
  ...
```

- Partition source into sections written and maintained by accelerator experts versus those evolved by science and engineering domain experts
Elements of an integrated model

- Execution model – host-directed execution with one or more attached accelerator devices
- Memory model – allow for separated host & device memories, an exposed device memory hierarchy
- Portable high-level programming with OpenACC directives
- Portable low-level programming based loosely on CUDA and OpenCL
New low-level procedure attributes

- Host proc’s in C/C++: `__host__ int func(...)`
  Fortran: `attributes(host) subroutine(...)

- Kernel proc’s in C/C++: `__kernel__ void func(...)`
  Fortran: `attributes(kernel) subroutine(...)

- Device proc’s in C/C++: `__device__ int func(...)`
  Fortran: `attributes(device) subroutine(...)`
New low-level variable attributes

- Host data (default)
- Device data in C/C++: __device__, __deviceptr__
  Fortran: attributes(device)
- Constant data in C/C++: __constant__
  Fortran: attributes(constant)
- Shared data in C/C++: __shared__
  Fortran: attributes(shared)
- Pinned data in C/C++: __pinned__
  Fortran: attributes(pinned)
Portable low-level API functions

- **Device mgmt** – acc_get_num_devices, acc_set_device_type, acc_get_device_type, acc_get_device_num, acc_set_device_num, acc_init, acc_shutdown

- **Memory mgmt** – acc_malloc, acc_free, acc_memset, acc_memset_async, acc_memcpy_to_device, acc_memcpy_to_device_async, acc_memcpy_to_host, acc_memcpy_to_host_async, ... error handling, etc

- **Asynchronous execution mgmt** – acc_async_test, acc_async_test_all, acc_async_wait, acc_async_wait_all, ... event mgmt?

... You get the idea
void mm_kernel(float *a, float *b, float *c, int n, int m, int l) {
    float cij;
    int i, j, kg, k, wx, wy;
    __shared__ float asub[16][17], bsub[16][17];
    wx = workerIdx.x;
    wy = workerIdx.y;
    i = gangIdx.x * 16 + wx;
    j = gangIdx.y * 16 + wy;
    cij = 0.0f;
    for (kg = 0; kg < m; kg += 16) {
        asub[wy][wx] = a[(kg+wy)*n+i];
        bsub[wy][wx] = b[j*l+(kg+ty)];
        syncworkers();
        for (k = 0; k < 16; k++)
            cij += asub[k][wx] * bsub[wy][k];
        syncworkers();
    }
    c[j*n+i] = cij;
}
Accelerator Language Landscape

Explicit

- CUDA
- C/C++

Implicit

- C++
- LEO

PGI Accelerator Language Landscape

- PGI CUDA Fortran
- Future OpenACC

PGI Accelerator

- 1.x
- 2.x
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.
Object-oriented features

Type extension allows polymorphism:

```fortran
type dertype
  integer  id, iop, npr
  real, allocatable :: rx(:)
contains
  procedure :: init => init_dertype
  procedure :: print => print_dertype
  procedure :: find  => find_dertype
end type dertype

type, extends(dertype) :: extdertype
  real, allocatable, device :: rx_d(:)
contains
  procedure :: init  => init_extdertype
  procedure :: find  => find_extdertype
end type extdertype
```

The class statement allows arguments of the base or extended type:

```fortran
subroutine init_dertype(this, n)
  class(dertype) :: this
end subroutine init_dertype
```
Single core scalar execution model

```cpp
for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
        f(i, j);
    }
}
```

...
Dual core parallel execution model

...< statement >
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<N; i++) {
        for (j=0; j<M; j++) {
            f(i, j);
        }
    }
}< statement >
...
OpenCL kernel execution model

```c
{
    int i;
    int j;

    i = get_global_id(0);
    j = get_global_id(1);
    f(i, j);
}
```
OpenCL kernel execution model on dual Cortex-A9

- Customizable Max WorkGroup size (256 Max)
- One pthread per CPU dedicated to manage WorkGroup execution
- 3 executions modes
  - Any WorkGroup size, no synchronization.
  - Any WorkGroup size, with synchronization. WorkItems are emulated by Light Weight Threads
  - WorkGroup size of 1 WorkItem with synchronization. Synchronization is ‘nop-ified’
Typical Porting Experience with PGI Accelerator OpenACC Directives
for (iter = 1; iter <= niter; ++iter)
{
    #pragma acc region
    {
        for (i = 1; i < n-1; ++i){
            for (j = 1; j < m-1; ++j){
                a[i][j]=w0*b[i][j]+ w1*(b[i-1][j]+b[i+1][j]+ b[i][j-1]+b[i][j+1])+ w2*(b[i-1][j-1]+b[i-1][j+1]+ b[i+1][j-1]+b[i+1][j+1]);
            }
        }
        for( i = 1; i < n-1; ++i )
        { for( j = 1; j < m-1; ++j )
            b[i][j] = a[i][j];
        }
    }
}
#pragma acc data region \
  copy(b[0:n-1][0:m-1]) \
  local(a[0:n-1][0:m-1])
{
  for (iter = 1; iter <= p; ++iter){
    #pragma acc region
    {
      for (i = 1; i < n-1; ++i){
        for (j = 1; j < m-1; ++j){
          a[i][j]=w0*b[i][j]+ w1*(b[i-1][j]+b[i+1][j]+ b[i][j-1]+b[i][j+1])+ w2*(b[i-1][j-1]+b[i-1][j+1]+ b[i+1][j-1]+b[i+1][j+1]);
        }
      }
    }
    for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
    b[i][j] = a[i][j];
  }
}