Programming Heterogeneous Parallel Architectures

Intermediate OpenACC

Michael Wolfe
Michael.Wolfe@pgroup.com
http://www.pgroup.com

CEA-INRIA-EDF

Ecoles d'Eté

2013

Outline

- cache directive
- inlining
- interfacing with CUDA
  - host_data
  - acc_malloc, acc_free, deviceptr
  - CUDA Fortran
- API calls
- OpenACC 2.0
- PGI-specific features

cache directive

```
#pragma acc parallel loop present(x[0:n],y[0:n])
    for( i = 1; i < n-1; ++i ){
        y[i] = 0.5*(x[i-1]*x[i+1]);
    }
```

cache directive

```
#pragma acc parallel loop present(x[0:n],y[0:n])
    for( i = 1; i < n-1; ++i ){
        #pragma acc cache(x[i-1:2])
        y[i] = 0.5*(x[i-1]*x[i+1]);
    }
```

- cross-iteration locality

cache directive

```
!$acc parallel loop present(x(1:n,1:m),y(1:n,1:m))
    do j = 2,m
      do i = 2,n
        x(i,j) = 0.25*(y(i-1,j) + y(i+1,j)+ &
                       y(i,j-1) + y(i,j+1))
      enddo
    enddo
```


cache directive

```
!$acc parallel loop present(x(1:n,1:m),y(1:n,1:m))
    do j = 2,m
      do i = 2,n
        !$acc cache(y(i-1:i+1,j-1:j+1))
        x(i,j) = 0.25*(y(i-1,j) + y(i+1,j)+ &
                       y(i,j-1) + y(i,j+1))
      enddo
    enddo
```

- cross-iteration locality

Procedure Calls in Compute Region

```
#pragma acc parallel loop present(x[0:n],y[0:n])
    for( i = 0; i < n; ++i )
        y[i] = foo( x[i] );


// where does ‘foo’ come from?

pgcc -Minline a.c
pgcc -Mextract=lib:mylib b.c
pgcc -Minline=lib:mylib a.c
pgcc -Minline=levels:10 a.c
pgcc -Minline=levels:2,foo,phoo,bar,lib:mylib a.c
```

Mixing OpenACC with CUDA 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 C

```
x = acc_malloc( 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
```

Mixing OpenACC with CUDA Fortran (PGI)

```
module mymod
  real, dimension(:), allocatable, device :: x
contains
 attributes(device) subroutine devproc(...)
  ...
 end subroutine
 subroutine hostproc(...)
  !$acc parallel
   do i = 1, n
     call devproc(a(i))        ! device call in same module
   enddo
  !$acc end parallel
 end subroutine
end module
```

Useful API calls

- `n = acc_get_num_devices( acc_device_nvidia )`
- `acc_set_device_num( 1, acc_device_nvidia )`
- `acc_async_test( n ), acc_async_test_all()`
- `acc_async_wait( n ), acc_async_wait_all()`
- `acc_init( acc_device_nvidia )`
- `if( acc_on_device( acc_device_nvidia ) )…`
- `ptr = acc_malloc( sizeof(int)*n )`
- `acc_copyin( x, sizeof(int)*n );`
  `call acc_copyin( x(1:n,1:m) )`

OpenACC 2.0 Highlights

- Procedure calls, separate compilation
- Nested parallelism
- Device-specific tuning, multiple devices
- Data management features and global data
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- atomic operations
- New API routines

Currently

```
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // must inline matvec
}
```

OpenACC 2.0

```
#pragma acc routine worker
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // procedure call on the device
}
```

OpenACC 2.0 routine

```
#pragma acc routine worker
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // procedure call on the device
}
```

```
#pragma acc routine worker
void matvec( float* v, float* x,
             float* a, int i,
             int n ){
  float xx = 0;
  #pragma acc loop reduction(+:xx)
  for( int j = 0; j < n; ++j )
    xx += a[i*n+j]*v[j];
  x[i] = xx;
}
```

OpenACC 2.0 routine bind

```
#pragma acc routine worker bind(mvdev)
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

```
void matvec( float* v, float* x,
             float* a, int i, int n ){
    float xx=0.0;

    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}

#pragma acc routine worker nohost
void mvdev( float* v, float* x,
            float* a, int i, int n ){
    float xx = 0.0;
    #pragma acc loop reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}
```

Nested Parallelism

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop …
for( int i = 0; i < n; ++i )
   matvec( v, x, i, n );
```

```
#pragma acc routine
matvec(…){
   #pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

Nested Parallelism

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel num_gangs(1)
{
   matvec( v0, x0, i, n );
   matvec( v1, x1, i, n );
   matvec( v2, x2, i, n );
}
```

```
#pragma acc routine
matvec(…){
   #pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

device_type(dev-type)

```
#pragma acc parallel loop num_gangs(200)


for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

device_type(dev-type)

```
#pragma acc parallel loop num_gangs(200)


for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

```
#pragma acc parallel loop \
  device_type(nvidia) num_gangs(200) …\
  device_type(radeon) num_gangs(400) …
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

Dynamic Data Lifetimes

```
void init( int n ){
…
#pragma acc enter data copyin( x[0:100] )
…
#pragma acc enter data present_or_create( y[0:n] )
…
}
```

Dynamic Data Lifetimes

```
void fini( int n ){
…
#pragma acc exit data delete( x[0:100] )
…
#pragma acc exit data copyout( y[0:n] )
…
}
```

Global Data

```
float x[1000];
#pragma acc declare create(x)
// static allocation, host + device

float y[1000];
#pragma acc declare device_resident(y)
// static allocation, device-only

float z[10000];
#pragma acc declare link(z)
// static allocation on host, dynamic allocation on device
```

Global Data

```
float z[10000];
#pragma acc declare link(z)
…

#pragma acc data copyin( z )
{// device z allocated and initialized here
   foo();
}// device z deallocated here
```

Global Data

```
module m
 real, allocatable :: x(:)
 !$acc declare create(x)
 ! x dynamically allocated on host+device

 real, allocatable :: y(:)
 !$acc declare device_resident(y)
 ! y dynamically allocated on device only

end module
```

Multiple Host Threads

- OpenACC 1.0 was thread agnostic
  - PGI 12.x implementation: each thread managed a context
  - Cray implementation: threads shared context
- OpenACC 2.0: all threads share same context on same device
  - Support for multiple threads sharing data on device
- OpenACC 2.0: more multiple device support
  - dynamically choose device
- Pitfalls!

Loop Directive

```
#pragma acc parallel
{
   #pragma acc loop gang/worker/vector/seq/auto
   for( i = 0; i < n; ++i )
     …
   #pragma acc loop gang/worker/vector/seq/auto
   for( i = 0; i < n; ++i )
      …
}
// gang outermost, worker middle, vector innermost
// auto may choose
```

Loop Directive

```
#pragma acc kernels
{
   #pragma acc loop gang/worker/vector/seq/auto
   for( i = 0; i < n; ++i )
     …
   #pragma acc loop gang/worker/vector/seq/auto
   for( i = 0; i < n; ++i )
      …
}
```

// gang outermost, worker middle, vector innermost

// auto may choose, auto is default in kernels

Loop Tiling

```
#pragma acc parallel
{
   #pragma acc loop tile(64,4) gang vector
   for( j = 1; j < n-1; ++j )
      for( i = 1; i < m-1; ++i )
         x[i][j] = 0.25*( y[i-1][j] + y[i+1][j] +
                          y[i][j-1] + y[i][j+1]);

}
// 4 is for outer loop, 64 is for next inner loop
// strip mines each loop: tile loop jt,it, element loops je,ie
// je has trip count 4, ie has trip count 64
// jt,it have mode gang, je,ie have mode vector
```

Loop Tiling

- may tile 2, 3, … loops
- tile loop may be gang, worker, seq
- element loop may be worker, seq, vector
- tile(*,*) means compiler chooses tile size

Async Additions

```
#pragma acc parallel async(5) wait(4,3)
{
   #pragma acc loop tile(64,4) gang vector
   for( j = 1; j < n-1; ++j )
      for( i = 1; i < m-1; ++i )
         x[i][j] = 0.25*( y[i-1][j] + y[i+1][j] +
                          y[i][j-1] + y[i][j+1]);

}
// this parallel region goes on queue 5
// this parallel region waits for existing events on queues 4,3
```

Async Additions

```
#pragma acc wait(3)
```
// this thread waits for queue 3 to complete

```
#pragma acc wait(3) async(5)
```

// queue 5 waits for all events on queue 3 to complete

atomic operations

```
#pragma acc parallel loop
   for( j = 1; j < n-1; ++j ){
       y = x[j];
       i = y & 0xf;
       #pragma acc atomic update
           ++bin[i];
   }
// essentially the OpenMP atomic operations
// atomic update, read, write, capture
// only native data lengths supported
```

New OpenACC 2.0 API calls

- ```
  acc_copyin( x, sizeof(int)*n );
  call acc_copyin( x(1:n,1:m) )
  ```
- ```
  acc_present_or_copyin( x, sizeof(int)*n )
  call acc_present_or_copyin( x(1:n,1:m) )
  ```
- ```
  acc_create( x, sizeof(int)*n );
  call acc_create( x(1:n,1:m) )
  ```
- ```
  acc_present_or_create( x, sizeof(int)*n )
  call acc_present_or_create( x(1:n,1:m) )
  ```
- ```
  acc_delete( x, sizeof(int)*n );
  call acc_delete ( x(1:n,1:m)
  ```
- ```
  acc_copyout( x, sizeof(int)*n );
  call acc_copyout( x(1:n,1:m)
  ```

New OpenACC 2.0 API calls

- `acc_update_device( x, sizeof(int)*n )`
  `call acc_update_device( x(1:n,1:m) )`
- `acc_update_self( x, sizeof(int)*n )`
  `call acc_update_self( x(1:n,1:m) )`
- `acc_map_data( devptr, hostptr, sizeof(float)*n )`
- `acc_unmap_data( hostptr )`
- `devptr = acc_deviceptr( hostptr )`
- `hostptr = acc_hostptr( devptr )`
- `acc_is_present( x, sizeof(float)*n )`
  `acc_is_present( a(1:n,1:m) )`
- `acc_memcpy_to_device( devptr, src, bytes )`
- `acc_memcpy_from_device( dest, devptr, bytes )`

Platform-Specific API calls

- `acc_get_current_cuda_device()`
- `acc_get_current_cuda_context()`
- `acc_get_cuda_stream( i )`
- `acc_set_cuda_stream( i, void* )`

PGI: multiple device support

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

PGI extension: deviceid

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

PGI extension: deviceid

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}

#pragma acc kernels deviceid(0)
{…}
```

PGI extension: host is a device

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
void foo( int devid ){
   #pragma acc kernels deviceid( devid )
   {…}
}
```

PGI extension: multiple device types

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_radeon,0);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

PGI extension: multiple device types

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_radeon,0);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

pgcc –acc –ta=nvidia,radeon,host foo.c

GPU Programming with CUDA (C and PGI CUDA Fortran) and the PGI Accelerator Programming Model

Michael Wolfe
Michael.Wolfe@pgroup.com
http://www.pgroup.com

March 2011

Part 4: Wrap-up

Accelerators are the Future of HPC

- HPC can't support its own designs
  - commodity parts, software as well as hardware
- Accelerators allow break from ISA compatibility
- Accelerators allow strong scaling
- GPUs are the game today
  - designs are essentially free
- You must program to the new model
- Downside
  - future of commodity may be in mobile
  - commodity cpus and accelerators may not solve HPC problems

Programming Accelerators

- **Goals**
  - **productivity**
  - **performance**
  - **portability**
- **Will it run fast on tomorrow's accelerators?**

Options

- **Low level**
  - **OpenCL, CUDA**
  - **full control**
  - **low on productivity, performance portability**
  - **high on performance**
  - **language is portable, even if programs are not**

Options

- **Libraries**
  - **Magma, etc.**
  - **programming to the library**
  - **essentially a limited-vocabulary language**
  - **high on portability, productivity, performance**
    - ***if* your program fits the vocabulary**

Options

- **Class library**
  - **TBB, Ct, ArBB, Thrust**
  - **A type system and implementation**
  - **Advantage: some information instantiated at compile time**
  - **Other advantages / disadvantages are the same as library approach**

Options

- **High level, PGI Accelerator model, (eventually OpenMP)**
  - **High on productivity, portability**
  - **Performance is improving over time**
  - **Open question: how portable is the model?**

- **[became OpenACC]**

How to Reach a Petaflop

- 10^6 = megaflop
- 10^9 = gigaflop
- 10^{12} = teraflop
- 10^{15} = petaflop
  - Jaguar
    - 18,688 dual-socket six-core nodes
      2.6GHz, 4-8 GFlops/core, 224,256 cores
    - (.224 x 10^6 cores) x (2.6 x 10^9 GHz) x (4 results)
      = 2.32 x 10^{15} results/cycle = 2.32 Petaflops (double precision)
    - Top500 Rmax = 1.759 PFlops, Rpeak = 2.331 PFlops

How to Reach an Exaflop

- 10^6 = megaflop
- 10^9 = gigaflop
- 10^{12} = teraflop
- 10^{15} = petaflop
- 10^{18} = exaflop = 10^9 x 10^9
  - one billion gigaflop cores = one exaflop (MPI^{max})
    - 16-cores in 2011, 32-2013, 64-2015, 128-2017, 256 in 2019
    - 1 million quad-socket 256-core nodes at 1GHz
    - 50X nodes, 2X sockets/node, 40X cores/socket relative to Jaguar
    - at 4 results/GHz, reduce by ¼, higher clock reduces as well
  - one million teraflop cores = one exaflop
    - 1,000 ops / cycle (1GHz)

How to Reach an Exaflop

- **Maybe O(100,000) 10-teraflop nodes**
  - **wide SIMD, multithreading, latency tolerant**
  - **1GHz clock = 10,000 operations/cycle (5,000 mul+add)**

Jaguar	Proposed
224,256 cores	O(100,000) units
37,376 sockets	100,000 sockets
18,688 nodes	O(100,000) nodes
2.5GHz clock	1GHz clock
4-8 GFlops/core	O(10,000) GFlops/unit
24-48 GFlops/socket	10,000 GFlops/socket
48-96 GFlops/node	O(10,000) GFlops/node

Top Machines 2013

	Jaguar	K	Sequoia	Titan	Tianhe-2
nodes	18688	88128	98304	18688	16000
CPUs	2*6	1*8	1*16	1*16	2*12
GHz	2.6	2.0	1.6	2.2	2.2
Ops/clk	4	8	8	4	8
gf/node	124	128	204	140	422
Pflops	2.3	11.2	20.1	2.6	6.7
ACCs				1*14	3*57
GHz				.73	1.1
Ops/clk				128	16
gf/acc				1311	3009
Pflops				24.5	48.1

Additional Information

- PGI Accelerator Programming Model
  - x86+NVIDIA
  - PGI Fortran and C
  - Linux, Windows, OSX
  - www.pgroup.com/accelerate for documentation, FAQ, articles
- PGI CUDA Fortran
  - X86 + NVIDIA
  - PGI Fortran
  - Linux, Windows, OSX
  - www.pgroup.com/cudafortran for documentation, FAQ, articles
- Common Compiler Feedback Format (CCFF)
  - integrated into all PGI compilers and pgprof
  - www.pgroup.com/CCFF for additional information

Copyright Notice

Where to get help

- PGI Customer Support - trs@pgroup.com
- PGI User's Forum - http://www.pgroup.com/userforum/index.php
- PGI Articles - http://www.pgroup.com/resources/articles.htm
  http://www.pgroup.com/resources/accel.htm
- PGI User's Guide - http://www.pgroup.com/doc/pgiug.pdf
- CUDA Fortran Reference Guide - http://www.pgroup.com/doc/pgicudafortug.pdf

Copyright Notice