

GPU Programming with the PGI Accelerator Programming Model

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

March 2011

Part 3: The PGI Accelerator Programming Model

3-1



Jacobi Relaxation

```
change = tolerance + 1.0
do while(change > tolerance)
  change = 0
  do i = 2, m-1
    do j = 2, n-1
      newa(i,j) = w0*a(i,j) + &
        w1 * (a(i-1,j) + a(i,j-1) + &
          a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
          a(i+1,j-1) + a(i+1,j+1))
      change = max(change,abs(newa(i,j)-a(i,j)))
    enddo
  enddo
  a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
enddo
```

3-3



Accelerator Programming

- Simple introductory program
- Programming model
- Building Accelerator programs
- High-level Programming
- Interpreting Compiler Feedback
- Tips and Techniques
- Performance Tuning

3-2



Jacobi Relaxation

```
do{
  change = 0;
  for( i = 1; i < m-1; ++i ){
    for( j = 1; j < n-1; ++j ){
      newa[j][i] = w0*a[j][i] +
        w1 * (a[j][i-1] + a[j-1][i] +
          a[j][i+1] + a[j+1][i]) +
        w2 * (a[j-1][i-1] + a[j+1][i-1] +
          a[j-1][i+1] + a[j+1][i+1]);
      change = fmax(change,fabs(newa[j][i]-a[j][i]));
    }
  }
  for( i = 1; i < m-1; ++i )
    for( j = 1; j < n-1; ++j )
      a[j][i] = newa[j][i];
}while( change > tolerance );
```

3-4



Jacobi Relaxation

```
change = tolerance + 1.0
!$omp parallel shared(change)
do while(change > tolerance)
  change = 0
  !$omp do reduction(max:change) private(i,j)
  do i = 2, m-1
    do j = 2, n-1
      newa(i,j) = w0*a(i,j) + &
        w1 * (a(i-1,j) + a(i,j-1) + &
          a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
          a(i+1,j-1) + a(i+1,j+1))
      change = max(change,abs(newa(i,j)-a(i,j)))
    enddo
  enddo
  a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
enddo
```

3-5

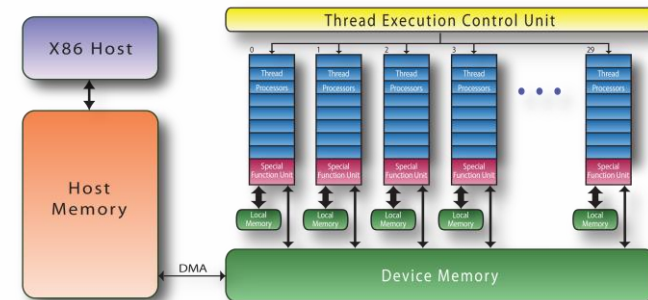
```
change = tolerance + 1.0
!$acc data region local(newa(1:m,1:n)) &
  copy(a(1:m,1:n))
do while(change > tolerance)
  change = 0
  !$acc region
  do i = 2, m-1
    do j = 2, n-1
      newa(i,j) = w0*a(i,j) + &
        w1 * (a(i-1,j) + a(i,j-1) + &
          a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
          a(i+1,j-1) + a(i+1,j+1))
      change = max(change,abs(newa(i,j)-a(i,j)))
    enddo
  enddo
  a(2:m-1,2:n-1) = newa(2:m-1,2:n-1)
!$acc end region
enddo
!$acc end data region
```

3-6

```
#pragma acc data region local(newa[0:n-1][0:m-1])\
  copy(a[0:n-1][0:m-1])
{ do{
  change = 0;
  #pragma acc region
  {
    for( i = 1; i < m-1; ++i )
      for( j = 1; j < n-1; ++j ){
        newa[j][i] = w0*a[j][i] +
          w1 * (a[j][i-1] + a[j-1][i] +
            a[j][i+1] + a[j+1][i]) +
          w2 * (a[j-1][i-1] + a[j+1][i-1] +
            a[j-1][i+1] + a[j+1][i+1]);
        change = fmax(change, fabs(newa[j][i]-a[j][i]));
      }
    for( i = 1; i < m-1; ++i )
      for( j = 1; j < n-1; ++j )
        a[j][i] = newa[j][i];
  }
}while( change > tolerance ); }
```

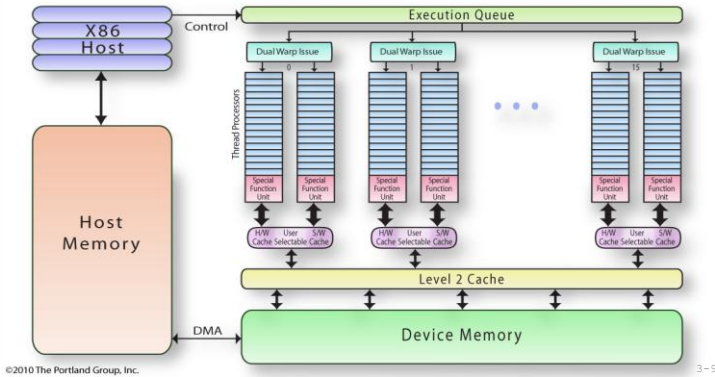
3-7

Abstracted x64+Tesla Accelerator Architecture



3-8

Abstracted x64+Fermi Accelerator Architecture



©2010 The Portland Group, Inc.

3-9

Why use Accelerator Directives?

- Productivity
 - Higher level programming model
 - a la OpenMP
- Portability
 - ignore directives, portable to the host
 - portable to other accelerators
 - performance portability
- Performance feedback
- Downsides
 - it's not as easy as inserting a few directives
 - a good host algorithm is not necessarily a good GPU algorithm

3-10

Basic Syntactic Concepts

- Fortran accelerator directive syntax
 - `!$acc directive [clause]...`
 - `&` continuation
 - Fortran-77 syntax rules
 - `!$acc` or `C$acc` or `*$acc` in columns 1-5
 - continuation with nonblank in column 6
- C accelerator directive syntax
 - `#pragma acc directive [clause]... eol`
 - continue to next line with backslash

3-11

Region

- region is single-entry/single-exit region
 - in Fortran, delimited by begin/end directives
 - in C, a single statement, or `{...}` region
 - no jumps into/out of region, no return
- compute region contains loops to send to GPU
 - loop iterations translated to GPU threads
 - loop indices become `threadidx/blockidx` indices
- data region encloses compute regions
 - data moved at region boundaries

3-12

Appropriate Algorithm

- Nested parallel loops
 - iterations map to threads
 - parallelism means threads are independent
 - nested loops means lots of parallelism
- Regular array indexing
 - allows for stride-1 array fetches

3-13

```
#pragma acc data region local(newa[0:n-1][0:m-1])\
    copy(a[0:n-1][0:m-1])
{ do{
  change = 0;
  #pragma acc region
  {
    for( i = 1; i < m-1; ++i )
      for( j = 1; j < n-1; ++j ){
        newa[j][i] = w0*a[j][i] +
          w1 * (a[j][i-1] + a[j-1][i] +
            a[j][i+1] + a[j+1][i]) +
          w2 * (a[j-1][i-1] + a[j+1][i-1] +
            a[j-1][i+1] + a[j+1][i+1]);
        change = fmax(change, fabs(newa[j][i]-a[j][i]));
      }
    for( i = 1; i < m-1; ++i )
      for( j = 1; j < n-1; ++j )
        a[j][i] = newa[j][i];
  }
}while( change > tolerance ); }
```

3-14

Behind the Scenes

- compiler determines parallelism
- compiler generates thread code
 - split up the iterations into threads, thread groups
 - inserts code to use software data cache
 - accumulate partial sum
 - second kernel to combine final sum
- compiler also inserts data movement
 - compiler or user determines what data to move
 - data moved at boundaries of data/compute region

3-15

Behind the Scenes

- virtualization penalties
 - fine grain control
 - thread scheduling
 - shared memory usage
 - loop unrolling

3-16

time for a live demo (1)

vector add

jacobi

3-17

Building Accelerator Programs

- ❑ `pgfortran -ta=nvidia a.f90`
- ❑ `pgcc -ta=nvidia a.c`
- ❑ **Other options:**
 - `-ta=nvidia[,cc10|cc11|cc12|cc13|cc20]`
 - default in `siterc` file:
 - set `COMPUTECAP=13;`
 - `-ta=nvidia[,cuda2.3|cuda3.1|cuda3.2]`
 - default in `siterc` file:
 - set `DEFCUDAVERSION=3.1;`
 - `-ta=nvidia,time`
 - `-ta=nvidia,host`
- ❑ Enable compiler feedback with `-Minfo` or `-Minfo=accel`

3-18

Performance Goals

- ❑ **Data movement between Host and Accelerator**
 - minimize amount of data
 - minimize number of data moves
 - minimize frequency of data moves
 - optimize data allocation in device memory
- ❑ **Parallelism on Accelerator**
 - Lots of MIMD parallelism to fill the multiprocessors
 - Lots of SIMD parallelism to fill cores on a multiprocessor
 - Lots more MIMD parallelism to fill multithreading parallelism

3-19

Performance Goals

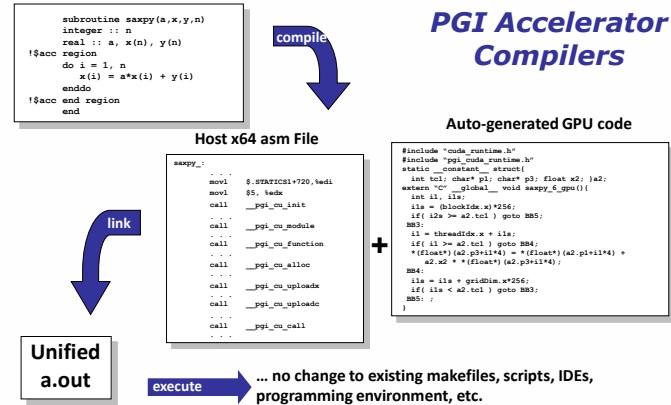
- ❑ **Data movement between device memory and cores**
 - minimize frequency of data movement
 - optimize strides – stride-1 in vector dimension
 - optimize alignment – 16-word aligned in vector dimension
 - store array blocks in data cache
- ❑ **Other goals?**
 - minimize register usage?
 - small kernels vs. large kernels?
 - minimize instruction count?
 - minimize synchronization points?

3-20

Program Execution Model

- Host
 - executes most of the program
 - allocates accelerator memory
 - initiates data copy from host memory to accelerator
 - sends kernel code to accelerator
 - queues kernels for execution on accelerator
 - waits for kernel completion
 - initiates data copy from accelerator to host memory
 - deallocates accelerator memory
- Accelerator
 - executes kernels, one after another
 - concurrently, may transfer data between host and accelerator

3-21



3-22

Compute Region

- C


```

#pragma acc region
{
    ....
}
            
```
- Fortran


```

!$acc region
    ....
!$acc end region
            
```

3-23

Compute Region

- C


```

#pragma acc region
{
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
            
```
- Fortran


```

!$acc region
    do i = 1,n
        r(i) = a(i) * 2.0
    enddo
!$acc end region
            
```

3-24

Compute Region Clauses

```

□ C
  #pragma acc region if(n > 100)
  {
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
  }

□ Fortran
  !$acc region if( n.gt.100 )
    do i = 1,n
      r(i) = a(i) * 2.0
    enddo
  !$acc end region

```

3-25

Compute Region Clauses

```

□ C
  #pragma acc region copyin(a)
  {
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
  }

□ Fortran
  !$acc region copyin(a)
    do i = 1,n
      r(i) = a(i) * 2.0
    enddo
  !$acc end region

```

3-26

Compute Region Clauses

```

□ C
  #pragma acc region copyin(a[0:n-1])
  {
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
  }

□ Fortran
  !$acc region copyin(a(1:n))
    do i = 1,n
      r(i) = a(i) * 2.0
    enddo
  !$acc end region

```

3-27

Compute Region Clauses

```

□ C
  #pragma acc region copyin(a[0:n-1]) copyout(r)
  {
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
  }

□ Fortran
  !$acc region copyin(a(1:n)) copyout(r)
    do i = 1,n
      r(i) = a(i) * 2.0
    enddo
  !$acc end region

```

3-28

Compute Region Clauses

```

❑ C
#pragma acc region copyin(a[0:n-1]) \
        copyout(r[0:n-1])
{
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}

❑ Fortran
!$acc region copyin(a(1:n)) copyout(r(1:n))
do i = 1,n
    r(i) = a(i) * 2.0
enddo
!$acc end region
    
```

3-29

Compute Region Clauses

```

❑ C
#pragma acc region copyin(a[0:n-1][0:m-1])
{
    for( i = 1; i < n-1; ++i )
        for( j = 1; j < m-1; ++j )
            r[i][j] = a[i][j]*2.0f;
}

❑ Fortran
!$acc region copyin(a(1:m,1:n))
do i = 2,n-1
do j = 2,m-1
    r(j,i) = a(j,i) * 2.0
enddo
enddo
!$acc end region
    
```

3-30

Compute Region Clauses

```

❑ C
#pragma acc region copyin(a[0:n-1][0:m-1]) \
        copy(r[0:n-1][0:m-1])
{
    for( i = 1; i < n-1; ++i )
        for( j = 1; j < m-1; ++j )
            r[i][j] = a[i][j]*2.0f;
}

❑ Fortran
!$acc region copyin(a(1:m,1:n)) copy(r(:,,:))
do i = 2,n-1
do j = 2,m-1
    r(j,i) = a(j,i) * 2.0
enddo
enddo
!$acc end region
    
```

3-31

Compute Region Clauses

```

!$acc region copyin(a(1:m,1:n)) local(r)
do times = 1,niters
do i = 2,n-1
do j = 2,m-1
    r(j,i) = 0.25*(a(j-1,i)+a(j,i-1)+a(j+1,i)+a(j,i+1))
enddo
enddo
do i = 2,n-1
do j = 2,m-1
    a(j,i) = r(j,i)
enddo
enddo
!$acc end region
    
```

3-32

Compute Region Clauses

```
!$acc region copyin(a(1:m,1:n)) local(r(2:m-1,2:n-1))
do times = 1, niters
do i = 2,n-1
do j = 2,m-1
do j = 2,m-1
r(j,i) = 0.25*(a(j-1,i)+a(j,i-1)+a(j+1,i)+a(j,i+1))
enddo
enddo
do i = 2,n-1
do j = 2,m-1
a(j,i) = r(j,i)
enddo
enddo
enddo
!$acc end region
```

3-33

Compute Region Clauses

```
□ C
#pragma acc region copyin(a[0:n-1][0:m-1])
/* data copied to Accelerator here */
{
for( i = 1; i < n-1; ++i )
for( j = 1; j < m-1; ++j )
r[i][j] = a[i][j]*2.0f;
} /* data copied to Host here */

□ Fortran
!$acc region copyin(a(1:m,1:n))
! data copied to Accelerator here
do i = 2,n-1
do j = 2,m-1
r(j,i) = a(j,i) * 2.0
enddo
enddo
! data copied to Host here
!$acc end region
```

3-34

Compute Region Clauses

- Conditional
 - `if(condition)`
- Data allocation clauses
 - `copy(list)`
 - `copyin(list)`
 - `copyout(list)`
 - `local(list)`
 - data in the lists must be distinct (data in only one list)
- Data update clauses
 - `updatein(list)` Or `update device(list)`
 - `updateout(list)` Or `update host(list)`
 - data must be in a data allocate clause for an enclosing data region

3-35

Compute Region Clauses

- `copyin` and `updatein` (host to gpu) at region entry
- `copyout` and `updateout` (gpu to host) at region exit

3-36

time for a live demo (2)
 compute region
 compute region clauses

3-37

Data Region

□ C

```
#pragma acc data region
{
    ....
}
```

□ Fortran

```
!$acc data region
....
!$acc end data region
```

□ May be nested and may contain compute regions

□ May not be nested within a compute region

3-38

Data Region Clauses

□ Data allocation clauses

- `copy(list)`
- `copyin(list)`
- `copyout(list)`
- `local(list)`
- data in the lists must be distinct (data in only one list)
- may not be in a data allocate clause for an enclosing data region

□ Data update clauses

- `updatein(list)` or `update device(list)`
- `updateout(list)` or `update host(list)`
- data must be in a data allocate clause for an enclosing data region

3-39

Data Region Update Directives

□ `update host(list)`

□ `update device(list)`

- data must be in a data allocate clause for an enclosing data region
- both may be on a single line
 - `update host(list) device(list)`

3-40

Passing Device Copies

```

subroutine sub( a, b )
  real :: a(:), b(:)
  !$acc reflected(a)
  !$acc region copyin(b)
  do i = 1,n
    a(i) = a(i) * b(i)
  enddo
  !$acc end region
  ...
end subroutine

subroutine bus(x, y)
  real :: x(:), y(:)
  !$acc data region copy(x)
  call sub( x, y )
  ...
end subroutine

subroutine sub( a, b )
  real, device :: a(:)
  real :: b(:)
  !$acc region copyin(b)
  do i = 1,n
    a(i) = a(i) * b(i)
  enddo
  !$acc end region
  ...
end subroutine

subroutine bus(x, y)
  real, device :: x(:)
  real :: y(:)
  call sub( x, y )
  ...
end subroutine

```

3-41



Passing Device Copies

- ❑ Reflected clause only available in Fortran
- ❑ List of argument arrays
- ❑ Caller must have a visible device copy at the call site
- ❑ Subprogram interface must be explicit
 - interface block or module
- ❑ Compiler will pass a hidden argument corresponding to the device copy

3-42

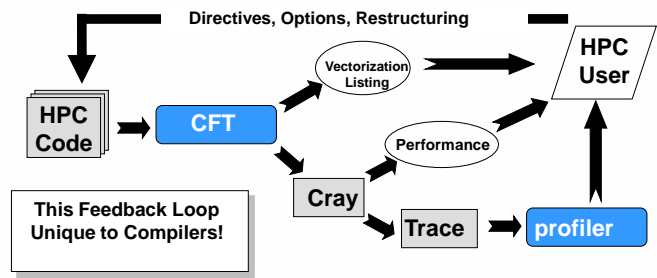


time for a live demo (3)
data region
compiler feedback

3-43



How did we make Vectors Work? Compiler-to-Programmer Feedback – a classic “Virtuous Cycle”

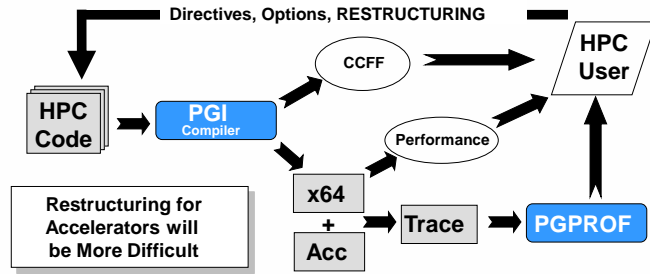


We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators

3-44



Compiler-to-Programmer Feedback



3-45

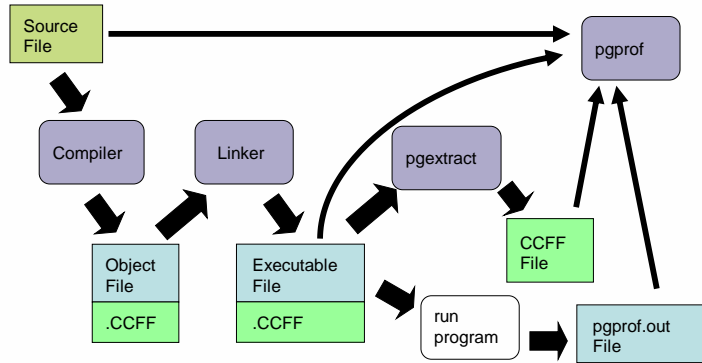
Compiler-to-User Feedback

```
% pgfortran -fast -ta=nvidia -Minfo mm.F90
mm1:
 6, Generating copyout(a(1:m,1:m))
  Generating copyin(c(1:m,1:m))
  Generating copyin(b(1:m,1:m))
 7, Loop is parallelizable
 8, Loop is parallelizable
  Accelerator kernel generated
 7, !$acc do parallel, vector(16)
 8, !$acc do parallel, vector(16)
11, Loop carried reuse of 'a' prevents parallelization
12, Loop is parallelizable
  Accelerator kernel generated
 7, !$acc do parallel, vector(16)
11, !$acc do seq
  Cached references to size [16x16] block of 'b'
  Cached references to size [16x16] block of 'c'
12, !$acc do parallel, vector(16)
  Using register for 'a'
```

3-46

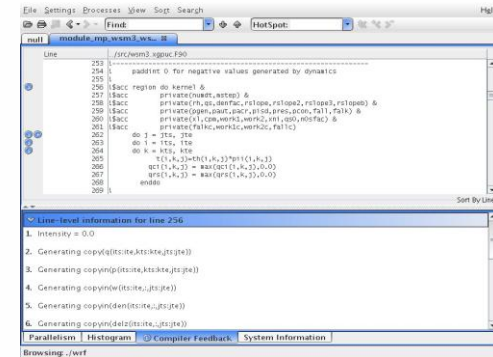
Common Compiler Feedback Format

<http://www.pgroup.com/ccff>



3-47

PGPROF with CCFF Messages



3-48

General Compiler Feedback

- ❑ How the function was compiled
- ❑ Interprocedural optimizations
- ❑ Profile-feedback runtime data
 - Block execution counts
 - Loop counts, range of counts
- ❑ Compiler optimizations, missed opportunities
 - Vectorization, parallelization
 - Altcode, re-ordering of loops, inlining
 - X64+GPU code generation, GPU kernel mapping, data movement
- ❑ Compute intensity – important for GPUs & Multi-core

3-49

Accelerator Compiler Feedback

- ❑ Generating `copyin(b(1:n,1:m))`
- ❑ Generating `copyout(b(2:n-1,2:m-1))`
- ❑ Generating `copy(a(1:n,1:n))`
- ❑ Generating `local(c(1:n,1:n))`
- ❑ Loop is parallelizable
- ❑ Accelerator kernel generated
- ❑ No parallel kernels found, accelerator region ignored
- ❑ Loop carried dependence due to exposed use of ... prevents parallelization
- ❑ Parallelization would require privatization of array ...

3-50

Accelerator Compiler Feedback

- ❑ hindrances
 - live out variables, non-private arrays
 - loop-carried dependences
 - unsupported datatype, unsupported operation
 - unknown array bounds
- ❑ data movement
 - `copyin`, `copyout`, `local`
- ❑ optimizations performed, not performance
 - cached data
 - non-stride-1 accesses (non-coalesced)
- ❑ loop schedule
 - mapping of loop iterations to thread/block indices

3-51

What can appear in a Compute Region?

- ❑ Arithmetic
 - C: int, float, double, struct
 - F: integer, real, double precision, complex, derived types
 - Loops, ifs
 - Kernel loops must be rectangular: trip count is invariant
- ❑ Obstacles with C
 - unbound pointers – use `restrict` keyword, or `-Msafeptr`, or `-Mipa=fast`
 - default is double – use float constants (`0.0f`), or `-Mfcon`, and float intrinsics
- ❑ Obstacles with Fortran
 - Fortran pointer attribute is not supported

3-52

C Intrinsic

□ C: #include <acclmath.h>

acos	asin	atan	atan2
cos	cosh	exp	fabs
fmax	fmin	log	log10
pow	sin	sinh	sqrt
tan	tanh		
acosf	asinf	atanf	atan2f
cosf	coshf	expf	fabsf
fmaxf	fminf	logf	log10f
powf	sinf	sinhf	sqrtf
tanf	tanhf		

3-53

Fortran Intrinsic

abs	acos	aint	asin
atan	atan2	cos	cosh
dble	exp	iand	ieor
int	ior	log	log10
max	min	mod	not
real	sign	sin	sinh
sqrt	tan	tanh	

3-54

other functions

- libm routines
 - use libm
 - #include <acclmath.h>
- device builtin routines
 - use cudadevice
 - #include <cudadevice.h>

3-55

Obstacles to Parallelization

- Unknown dependences
 - use simple array subscripts
- Unstructured Accumulations
 - rewrite to accumulate to local element
- Scalars whose last value is needed after the loop
 - declare local, or private, or rename in the loop
 - global variables, Fortran dummy arguments, are always live-out

3-56

Unstructured Accumulation

```
do i = 1, n
  x = someveryexpensivecomputation(i)
  ni = nbridx(i)
  do nbr = 1, nbrcnt(i)
    j = nbr(ni+nbr)
    flow(j) = flow(j) + x
  enddo
enddo
```

3-57

Restructured Accumulation

```
do i = 1, n
  x(i) = someveryexpensivecomputation(i)
enddo
do j = 1, n
  nj = invnbridx(i)
  do nbr = 1, invnbrcnt(j)
    i = invnbr(nj+nbr)
    flow(j) = flow(j) + x(i)
  enddo
enddo
```

3-58

Loop Schedules

```
27, Accelerator kernel generated
26, !$acc do parallel, vector(16)
27, !$acc do parallel, vector(16)
```

- vector loops correspond to threadidx indices
- parallel loops correspond to blockidx indices
- this schedule has a CUDA schedule
<<< dim3(ceil(N/16),ceil(M/16)), dim3(16,16) >>>
- Compiler strip-mines to protect against very long loop limits

3-59

Common Loop Schedules

- parallel
- vector
- vector(n)
- parallel, vector(n)
- seq
- host

3-60

Loop Directive

□ C

```
#pragma acc for clause...
for( i = 0; i < n; ++i ){
    ....
}
```

□ Fortran

```
!$acc do clause...
do i = 1, n
```

3-61

Loop Scheduling Clauses

□ !\$acc do parallel

- runs in 'parallel' mode only (blockIdx)
- does not declare that the loop is in fact parallel (use independent)

□ !\$acc do parallel(32)

- runs in 'parallel' mode only with gridDim == 32 (32 blocks)

□ !\$acc do vector(128)

- runs in 'vector' mode (threadIdx) with blockDim == 128 (128 threads)
- vector size, if present, must be compile-time constant

□ !\$acc do parallel vector(128)

- strip mines loop
- inner loop runs in vector mode, 128 threads (threadIdx)
- outer loop runs across thread blocks (blockIdx)

3-62

Loop Scheduling Clauses

- Want stride-1 loop to be in 'vector' mode (threadIdx)
 - look at -Minfo messages!
- Want lots of parallelism

3-63

Kernel Clause

```
!$acc region
do i = 1, n
do j = 2, m-1
    a(i,j) = 0.5*(b(i,j-1) + b(i,j+1))
enddo
do j = 2, m-1
    b(i,j) = a(i,j)
enddo
enddo
!$acc end region
```

3-64

Kernel Clause

```
!$acc region
!$acc do vector
do i = 1, n
!$acc do parallel
do j = 2, m-1
a(i,j) = 0.5*(b(i,j-1) + b(i,j+1))
enddo
enddo
!$acc do vector
do i = 1, n
!$acc do parallel
do j = 2, m-1
b(i,j) = a(i,j)
enddo
enddo
enddo
!$acc end region
```

3-65

Kernel Clause

```
!$acc region
!$acc do parallel vector(128)
do i = 1, n
!$acc do seq
do j = 2, m-1
a(i,j) = 0.5*(b(i,j-1) + b(i,j+1))
enddo
!$acc do seq
do j = 2, m-1
b(i,j) = a(i,j)
enddo
enddo
!$acc end region
```

3-66

Kernel Clause

```
!$acc region
!$acc do parallel vector(128) kernel
do i = 1, n
do j = 2, m-1
a(i,j) = 0.5*(b(i,j-1) + b(i,j+1))
enddo
do j = 2, m-1
b(i,j) = a(i,j)
enddo
enddo
!$acc end region
```

3-67

Loop Directive Clauses

- Scheduling Clauses
 - vector or vector(n)
 - parallel or parallel(n)
 - seq or seq(n)
 - host or host(n)
 - unroll(n) (after parallel, vector, seq)
- kernel
 - useful when you want a large outer loop body to be the kernel
- independent
 - use with care, overrides compiler analysis for dependence, private variables
- private(list)
 - private data for each iteration of the loop
 - different from local (how?)

3-68

time for a live demo (4)

loop clauses

3-69

Compiler Feedback Messages

□ Data related

- Generating copyin(b(1:n,1:m))
- Generating copyout(b(2:n-1,2:m-1))
- Generating copy(a(1:n,1:n))
- Generating local(c(1:n,1:n))

□ Loop or kernel related

- Loop is parallelizable
- Accelerator kernel generated

□ Barriers to GPU code generation

- No parallel kernels found, accelerator region ignored
- Loop carried dependence due to exposed use of ... prevents parallelization
- Parallelization would require privatization of array ...

Compiler Messages Continued

□ Memory optimization related

- Cached references to size [18x18] block of 'b'
- Non-stride-1 memory accesses for 'a'

Time for a Live Demo

□ Himeno benchmark

- Effects of using clauses on parallel schedule / performance
- Effects of padding

Obstacles to Parallelization

- Computed Index (linearization, look-up)
- While Loops
- Triangular Loops
- “live-out” Variables
- Privatization of Local Arrays
- Function calls
- Device Runtime Errors
- Compiler Errors



Computed Index – Linearization

```
!$acc region
do i = 1, M
  do j = 1, N
    idx = ((i-1)*M)+j
    A(idx) = B(i,j)
  enddo
enddo
!$acc end region
```

```
pgf90 linearization.f90 -ta=nvidia -Minfo=accel
linear:
16, No parallel kernels found, accelerator region ignored
17, Complex loop carried dependence of 'a' prevents parallelization
18, Complex loop carried dependence of 'a' prevents parallelization
Parallelization would require privatization of array 'a(:)
```

To fix,
Remove the linearization or Use the “independent” clause

```
!$acc region
do i = 1, M
  do j = 1, N
    A(i,j) = B(i,j)
  enddo
enddo
!$acc end region
```

```
!$acc region
!$acc do independent
do i = 1, M
  do j = 1, N
    idx = ((i-1)*M)+j
    A(idx) = B(i,j)
  enddo
enddo
!$acc end region
```



Computed Index - Look-up

```
!$acc region
do i = 1, M
  idx = lookup(i)
  do j = 1, N
    A(idx,j) = ((i-1)*M)+j
  enddo
enddo
!$acc end region
```

```
% pgf90 lookup.f90 -ta=nvidia -Minfo=accel
lookup_test:
16, Generating copyout(a(:,1:1024))
17, Parallelization would require privatization of array 'a(:,1:1024)'
Sequential loop scheduled on host
19, Loop is parallelizable
Accelerator kernel generated
19, !$acc do parallel, vector(256)
```

```
!$acc region
do i = 1, M
  do j = 1, N
    idx = lookup(i)
    A(i,idx) = ((i-1)*M)+j
  enddo
!$acc end region
```

```
% pgf90 lookup1.f90 -ta=nvidia -Minfo=accel
lookup_test:
16, Generating copyout(a(1:1024,:))
Generating copyin(cell(1:1024))
17, Loop is parallelizable
Accelerator kernel generated
17, !$acc do parallel, vector(256)
18, Loop carried reuse of 'a' prevents parallelization
Inner sequential loop scheduled on accelerator
```

The Independent or parallel clauses could be used to force parallelization but is not recommended



While Loops

```
!$acc region
i = 0
do, while (.not.found)
  i = i + 1
  if (A(i) .eq. 102) then
    found = i
  endif
enddo
!$acc end region
```

```
% pgf90 -ta=nvidia -Minfo=accel while1.f90
while1:
17, Accelerator region ignored
19, Accelerator restriction: invalid loop
```

Change to a rectangular loop

```
!$acc region
do i = 1, N
  if (A(i) .eq. 102) then
    found(i) = i
  else
    found(i) = 0
  endif
enddo
!$acc end region
print *, 'Found at ', maxval(found)
```

```
% pgf90 -ta=nvidia -Minfo=accel while2.f90
while2:
18, Generating copyin(a(1:1024))
Generating copyout(found(1:1024))
Generating compute capability 1.0 binary
Generating compute capability 1.3 binary
19, Loop is parallelizable
Accelerator kernel generated
19, !$acc do parallel, vector(256)
Using register for 'found'
```



Triangular Loops

```
!$acc region copyout(A)
do i = 1, M
  do j = i, N
    A(i,j) = i+j
  enddo
enddo
!$acc end region
```

All loop schedules must be rectangular. For triangular loops, the compiler will either serialize the inner loop or make the inner loop rectangular and add an implicit if statement to skip the lower part of the triangle.

Problem: The compiler will copy out the entire array A. The lower triangle contains garbage since it was not initialized. Use "copy(A)" to initialize the values.



"live-out" Variables

```
!$acc region
do i = 1, M
  do j = 1, N
    idx = i+j
    A(i,j) = idx
  enddo
enddo
!$acc end region
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel liveout.f90
liveout:
11, Generating copyout(a(1:1024,1:1024))
12, Loop is parallelizable
Accelerator kernel generated
12, !$acc do parallel, vector(256)
13, Inner sequential loop scheduled on accelerator
14, Accelerator restriction: induction variable live-out from loop: idx
15, Accelerator restriction: induction variable live-out from loop: idx
```

Privatize the scalar

```
!$acc region
do i = 1, M
!$acc do private(idx)
  do j = 1, N
    idx = i+j
    A(i,j) = idx
  enddo
enddo
!$acc end region
print *, idx, A(1,1), A(M,N)
```

```
% pgf90 -ta=nvidia,time -Minfo=accel liveout2.f90
liveout2:
10, Generating copyout(a(1:1024,1:1024))
11, Loop is parallelizable
13, Loop is parallelizable
Accelerator kernel generated
11, !$acc do parallel, vector(16)
13, !$acc do parallel, vector(16)
```



Privatization of Local Arrays

```
!$acc region
do i = 1, M
  do j = 1, N
    do jj = 1, 10
      tmp(jj) = jj
    end do
    A(i,j) = sum(tmp)
  enddo
enddo
!$acc end region
```

```
% pgf90 -ta=nvidia -Minfo=accel private.f90
privatearr:
10, Generating copyout(tmp(1:10))
Generating compute capability 1.0 binary
Generating compute capability 1.3 binary
11, Parallelization would require privatization of array 'tmp(1:10)'
13, Parallelization would require privatization of array 'tmp(1:10)'
Sequential loop scheduled on host
14, Loop is parallelizable
Accelerator kernel generated
14, !$acc do parallel, vector(10)
17, Loop is parallelizable
Accelerator kernel generated
17, !$acc do parallel, vector(10)
Sum reduction generated for tmp$r
```



Privatization of Local Arrays - cont.

```
!$acc region
do i = 1, M
!$acc do private(tmp)
  do j = 1, N
    do jj = 1, 10
      tmp(jj) = jj
    end do
    A(i,j) = sum(tmp)
  enddo
enddo
!$acc end region
```

```
% pgf90 -ta=nvidia,time -Minfo=accel private2.f90
privatearr2:
10, Generating copyout(a(1:1024,1:1024))
Generating compute capability 1.0 binary
Generating compute capability 1.3 binary
11, Loop is parallelizable
13, Loop is vectorizable
Accelerator kernel generated
11, !$acc do parallel, vector(16)
13, !$acc do vector(16)
14, Loop is parallelizable
17, Loop is parallelizable
```



Need to privatize local temporary arrays. Default is to assume that they are shared.

Function Calls

- Function calls are not allowed within a compute region.
- Restriction is due to lack of a device linker and hardware support.
- Functions must be inlined, either manually or by the compiler with `-Minline` or `-Mipa=inline`.



Device Errors

call to cuMemcpyDtoH returned error 700: Launch failed

```
!$acc region
do i = 1, M
  do j = 1, N
    A(i,j) = B(i,j+1) << out-of-bounds
  enddo
enddo
!$acc end region
```

- Typically occurs when the device kernel gets an execution error, such as a out-of-bounds or other memory access violation.

call to cuMemcpy2D returned error 1: Invalid value

```
parameter(N=1024,M=512)
real :: A(M,N), B(M,N)
...
!$acc region copyout(A), copyin(B(0:N,1:M+1)) <<< Bad bounds for the copyin
do i = 1, M
  do j = 1, N
    A(i,j) = B(i,j+1)
  enddo
enddo
!$acc end region
```

- Occurs if there is an error when copying data to/from the device



Compiler Errors

- No software is without bugs, including the compiler
- If you encounter a problem that seems to be the fault of the compiler, please send a report to PGI Customer Service (trs@pgroup.com)

Example of an Internal Compiler Error (ICE)

```
% pgf90 -ta=nvdiia -c bug.f90
/tmp/pgaccrVcZyn0dlyP.gpu(20): error: identifier "z0" is undefined

1 error detected in the compilation of "/tmp/pgnvdASVco8Pmox9p.nv0".
PGF90-F-0000-Internal compiler error. pgnvd job exited with nonzero status code 0 (bug.f90: 22)
```



Combined Directives

```
□ C
#pragma acc region for copyin(a[0:n-1])\
    parallel vector(32)
for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;

□ Fortran
!$acc region do copyin(a(1:n)) parallel vector(32)
do i = 1,n
  r(i) = a(i) * 2.0
enddo
```



Selecting Device

□ C

```
#include <accel.h>
n = acc_get_num_devices(acc_device_nvidia);
...
acc_set_device_num( 1, acc_device_nvidia);
```

□ Fortran

```
use accel_lib
n = acc_get_num_devices( acc_device_nvidia )
...
call acc_set_device_num( 0, acc_device_nvidia)
```

□ Environment Variable

```
setenv ACC_DEVICE_NUM 1
export ACC_DEVICE_NUM=1
```

3-85

PGI Unified Binary

□ One binary that executes on GPU or on host

- default: auto-detect presence of GPU, use GPU if present
- override default with environment variable ACC_DEVICE
- override within program with call to acc_set_device

□ Building PGI Unified Binary

- pgfortran -ta=nvidia,host -fast program.f90

□ Running PGI Unified Binary

- a.out
- setenv ACC_DEVICE host ; a.out
- setenv ACC_DEVICE nvidia ; a.out
- before first region
 - call acc_set_device(acc_device_nvidia)
 - call acc_set_device(acc_device_host)

3-86

PGI Unified Binary Interactions

□ PGI Unified Binary for multiple host types:

- pgfortran -tp=barcelona-64,core2-64
- up to H optimized copies of each routine with H host types

□ PGI Unified Binary for multiple accelerator types:

- pgfortran -ta=nvidia,host
- up to two optimized copies of each routine (with accelerator regions)

□ PGI Unified Binary for both multiple host and accelerator types:

- pgfortran -tp=barcelona-64,core2-64 -ta=nvidia,host
- with H host types and A accelerator types:
 - up to H optimized copies for each of H hosts, plus
 - up to A accelerator-enabled copies, optimized for “generic” host

3-87

time for a live demo (5)

selecting device

PGI Unified Binary

3-88

Performance Profiling

- ❑ TAU (Tuning and Analysis Utilities, University of Oregon)
 - collects performance information
- ❑ cudaprof (NVIDIA)
 - gives a trace of kernel execution
- ❑ `pgfortran -ta=nvidia,time` (on link line)
 - dump of region-level and kernel-level performance
 - upload/download data movement time
 - kernel execution time
- ❑ `pgcollect a.out`
- ❑ `PGI_ACC_PROFILE` environment variable
 - enables profile data collection for accelerator regions
- ❑ `ACC_NOTIFY` environment variable
 - prints one line for each kernel invocation

3-89

Performance Profiling

```
Accelerator Kernel Timing data
f3.f90
smooth
  24: region entered 1 time
      time(us): total=1116701 init=1115986 region=715
              kernels=22 data=693
w/o init: total=715 max=715 min=715 avg=715
  27: kernel launched 5 times
      grid: [7x7] block: [16x16]
      time(us): total=17 max=10 min=1 avg=3
  34: kernel launched 5 times
      grid: [7x7] block: [16x16]
      time(us): total=5 max=1 min=1 avg=1
```

3-90

Profiling an Accelerator Model Program

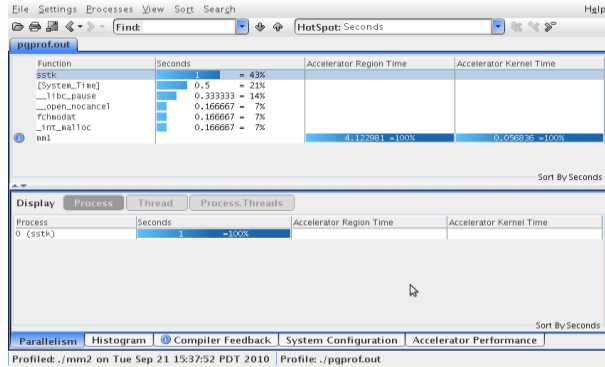
```
$ make
pgf90 -fast -Minfo=ccff ../mm2.f90 ../mmdriv.f90 -o mm2 -ta=nvidia
../mm2.f90:
../mmdriv.f90:
$
$ pgcollect -time ./mm2 < ../in
[...program output...]
target process has terminated, writing profile data
$
$ pgprof -exe ./mm2
```

- Build as usual
 - Here we add `-Minfo=ccff` to enhance profile data
- Just invoke with `pgcollect`
 - Currently collects data similar to `-ta=time`
- Invoke the `PGPROF` performance profiler

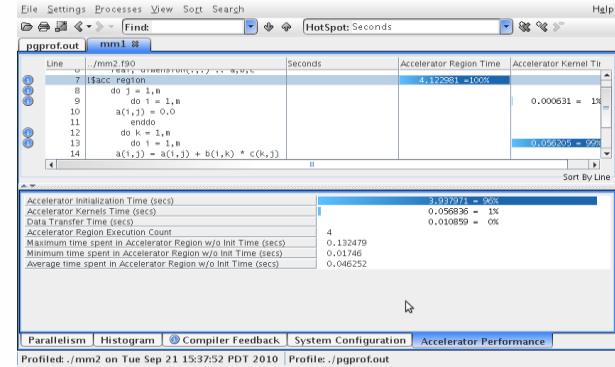
Dynamic Profile Data Collection (11.4)

- `acc_enable_time(acc_device_nvidia);`
...
`long etime = acc_exec_time(acc_device_nvidia);`
`long ttime = acc_total_time(acc_device_nvidia);`
`acc_disable_time(acc_device_nvidia);`
- `unsigned int allocs = acc_allocs();`
`unsigned int frees = acc_frees();`
`unsigned int copyins = acc_copyins();`
`unsigned int copyouts = acc_copyouts();`
`unsigned long bytesalloc = acc_bytesalloc();`
`unsigned long bytesin = acc_bytesin();`
`unsigned long bytesout = acc_bytesout();`
`unsigned int kernels = acc_kernels();`
`unsigned int regions = acc_regions();`
`unsigned long totmem = acc_get_memory();`
`unsigned long freemem = acc_get_free_memory();`

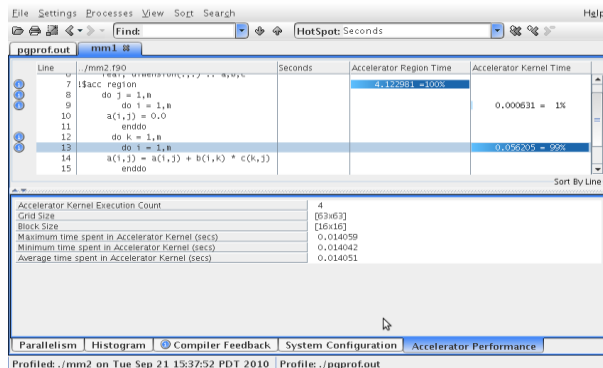
Accelerator Model Profiling



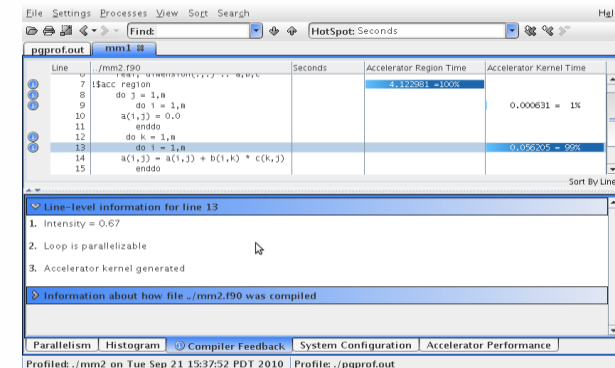
Accelerator Profiling - Region



Accelerator Profiling - Kernel



Compiler Feedback



time for a live demo (6)

cudaprof

-ta=nvidia,time

pgcollect and **pgprof**

ACC_NOTIFY

3-97

Behind the Scenes

- ❑ Live variable analysis and Array region analysis
 - augments information in region / loop directive clauses
 - determines in/out sets for the region
 - programmer feedback
- ❑ Dependence and parallelism analysis
 - determine loops that can be executed in parallel
 - programmer feedback
- ❑ Map loops onto hardware parallelism
 - parallel / vector (MIMD/SIMD), including strip mining
 - programmer feedback
- ❑ Kernel extraction, GPU code generation
 - kernel code optimization
 - same code generator as CUDA Fortran
- ❑ Host code generation
 - allocate memory, copy data, upload kernel code, launch kernel, copy results, deallocate memory
 - for unified binary, keep a host version of the loop

3-98

Directives Summary

- ❑ **acc region**
 - `if(condition)`
 - `copy(list) copyin(list) copyout(list) local(list)`
 - `updatein(list) updateout(list)`
- ❑ **acc data region**
 - `copy(list) copyin(list) copyout(list) local(list)`
 - `updatein(list) updateout(list)`
- ❑ **acc do Or acc for**
 - `host(width) parallel(width) seq(width) vector(width)`
 - `unroll(width)`
 - `kernel`
 - `independent`
 - `private(list)`

3-99

Directives Summary

- ❑ `update host(list)`
- ❑ `update device(list)`
- ❑ `_ACCEL` macro


```

#ifdef _ACCEL
    call acc_init( acc_device_nvidia )
#endif
            
```

3-100

Runtime Summary

- ❑ `acc_get_num_devices(acc_device_nvidia)`
- ❑ `acc_set_device_num(n, acc_device_nvidia)`
- ❑ `acc_set_device(acc_device_nvidia | acc_device_host)`
- ❑ `acc_get_device()`
- ❑ `acc_init(acc_device_nvidia | acc_device_host)`

3-101

Environment Variable Summary

- ❑ `ACC_DEVICE_NUM`
- ❑ `ACC_DEVICE`
- ❑ `ACC_NOTIFY`

3-102

Command Line Summary

- ❑ `-ta=nvidia`
- ❑ `-ta=nvidia,time`
- ❑ `-ta=nvidia,nofma`
- ❑ `-ta=nvidia,fastmath`
- ❑ `-ta=nvidia,mul24` (Tesla only)
- ❑ `-ta=nvidia,[cc10|cc11|cc12|cc13|cc20]` (multiple allowed)
- ❑ `-ta=nvidia,maxregcount:n`
- ❑ `-ta=nvidia,host`

3-103

Multiple Devices

- ❑ Nest Accelerator Region in OpenMP Parallel
- ❑ Each thread chooses one NVIDIA device
- ❑ Each thread computes its own loop limits
 - Not the same as `!$omp for`
- ❑ One GPU per host thread (CUDA limitation)
- ❑ All threads/GPUs share PCI bus

3-104

Multiple Devices

```
!$omp parallel private(ilo,ih,i) &
    num_threads(2)
    call acc_set_device(omp_get_thread_num())
    ilo = omp_get_thread_num()*(N+1)/2 + 1
    ihi = min(N,ilo+(N+1)/2)
    !$acc region do
    do i = ilo,ih,i
        a(i) = b(i) + c(i)
    enddo
!$omp end region
```

3-105

Performance Tuning

- ❑ Performance Measurement
- ❑ Choose an appropriately parallel algorithm
- ❑ Optimize data movement between host and GPU
 - frequency, volume, regularity
- ❑ Optimize device memory accesses
 - strides, alignment
 - use data cache
- ❑ Optimize kernel code
- ❑ Optimize compute intensity
- ❑ Linux only: pccudainit
- ❑ Windows: must be on the console
- ❑ OSX: two GPUs doesn't mean you can use the idle one

3-106

Writing Accelerator Programs

- ❑ Step 1: Appropriate algorithm: lots of parallelism
 - Lots of MIMD parallelism to fill the multiprocessors
 - Lots of SIMD parallelism to fill cores on a multiprocessor
 - Lots more MIMD parallelism to fill multithreading parallelism
 - High compute intensity
 - Insert directives, read feedback for parallelism hindrances
 - Dependence relations, pointers
 - Scalars live out from the loop
 - Arrays that need to be private
 - Iterate

3-107

Writing Accelerator Programs

- ❑ Step 2: Tune data movement between Host and Accelerator
 - read compiler feedback about data moves
 - minimize amount of data moved
 - minimize frequency of data moves
 - minimize noncontiguous data moves
 - optimize data allocation in device memory
 - optimize allocation in host memory (esp. for C)
 - insert local, copyin, copyout clauses
 - use data regions, update clauses

3-108

Manually managing memory (CUDA)

- CUDA Fortran device arrays
 - CUDA Fortran device arrays can be used in accelerator regions
 - compile with `.cuf` suffix or `-Mcuda` in addition to `-ta=nvidia`
- CUDA C device arrays
 - no attribute to tell the compiler where a pointer points
 - new in PGI 11.4
 - `#pragma acc [data] region deviceptr(ap)`
 - tells the compiler that `ap` was allocated on the GPU
 - `ap = acc_malloc(nbytes); ... acc_free(ap);`
 - allocates / frees `ap` on the current device
 - You may also use data allocated by `cudaMalloc` or `cuMemAlloc`

3-109

Tuning Accelerator Programs

- Step 3: Tune data movement between device memory and cores
 - minimize frequency of data movement
 - optimize strides – stride-1 in vector dimension
 - optimize alignment – 16-word aligned in vector dimension
 - look at cache feedback from compiler

3-110

Tuning Accelerator Programs

- Step 4: Tune kernel code
 - profile the code (`cuda_prof`, `pgcollect`, `-ta=nvidia,time`)
 - experiment with kernel schedule using loop directives
 - unroll clauses
 - enable experimental optimizations (`-ta=nvidia,O3`)
 - single precision vs. double precision
 - 24-bit multiply for indexing (`-ta=nvidia,mul24`)
 - low precision transcendentals (`-ta=nvidia,fastmath`)

3-111

unroll clauses

- Consider each vector space as a 'tile' of iterations
- `!$acc do parallel unroll(2)`
 - each thread block does two tiles
 - `!$acc do parallel unroll(4)` does four tiles
- `!$acc do vector(128) unroll(2)`
 - tile size is 128 in this dimension
 - use thread block of ½ tile size (64) to do this tile
 - `!$acc do vector(128) unroll(4)` uses ¼ tile size (32)
- `!$acc do parallel unroll(2) vector(128) unroll(4)`
 - strip mines loop to tile size 128
 - each thread block does two tiles
 - thread block size is (32 in this dimension)
- unroll width must be compile-time constant

3-112

Near Term Future Additions

- ❑ passing device data to subroutines (available now in Fortran)
- ❑ more code optimization
 - aggressive procedure inlining in compute regions
 - instruction count optimizations
 - post-optimal low-level code improvements
- ❑ automatic private array detection
- ❑ improvements to kernel schedule selection
- ❑ Direct calling of CUDA routines inside region
- ❑ Asynchronous data movement
- ❑ Asynchronous compute regions

3-113

The Portland Group®

Longer Term Evolution

- ❑ C++
- ❑ New targets
 - multicore, Larrabee?, ATI?, other?
- ❑ Overlap compute / data movement
 - pipelined loops
- ❑ More tools support
- ❑ Libraries of device routines
- ❑ Programming model evolution
 - multiple GPUs
 - standardization
 - Concurrent kernels, a la OpenCL / Fermi

3-114

The Portland Group®

PGI Accelerator vs CUDA/OpenCL

- ❑ The PGI Accelerator programming model is a high-level *implicit* programming model for x64+GPU systems, similar to OpenMP for multi-core x64. The PGI Accelerator model:
 - Enables offloading of compute-intensive loops and code regions from a host CPU to a GPU accelerator using simple compiler directives
 - Implements directives as Fortran comments and C pragmas, so programs can remain 100% standard-compliant and portable
 - Makes GPGPU programming and optimization incremental and accessible to application domain experts
 - Is supported in both the PGF95 and PGCC C99 compilers

3-115

The Portland Group®

PGI Accelerator vs CUDA/OpenCL

- ❑ CUDA is a lower-level explicit programming model with substantial runtime library components that give expert programmers direct control of:
 - Splitting up of source code into host CPU code and GPU compute kernel code in appropriately defined functions and subprograms
 - Allocation of page-locked host memory, GPU device main memory, GPU constant memory and GPU shared memory
 - All data movement between host main memory and the various types of GPU memory
 - Definition of thread/block grids and launching of compute kernels
 - Synchronization of threads within a CUDA thread group
 - Asynchronous launch of GPU compute kernels, synchronization with host CPU
- ❑ OpenCL is similar to CUDA, adds some functionality, even lower level

3-116

The Portland Group®

Why Not Just Use OpenMP?

- ❑ **Caveat:** PGI is an active member of the OpenMP ARB and language committee
 - OpenMP is looking at expanding its model to GPUs starting with the PGI Accelerator model
- ❑ **OpenMP model**
 - shared memory with weak memory model
 - long-lived threads + work sharing
 - defined synchronization points
 - dynamic scheduling, nested parallelism
 - model is prescriptive, not descriptive

3-117

Why Not Just Use OpenMP?

- ❑ **Preserve all of OpenMP**
 - simulate OpenMP thread behavior on GPU
 - simulate scheduling, synchronization, critical sections
- ❑ **Only allow a subset of OpenMP**
 - define a subset that maps well
 - disallow dynamic schedules, synchronization
- ❑ **Break OpenMP, define new behavior**
 - perhaps define master thread as the host, worker threads as on the GPU

3-118

OpenMP Accelerator Subcommittee

- ❑ `!$omp acc_region [clause]...`
`#pragma omp acc_region [clause]...`
 - `acc_shared(list)`
 - `acc_copy(list)`
 - `acc_copyin(list)`
 - `acc_copyout(list)`
 - `private(list)`
 - `present(list)`
 - `num_pes(depth:num)`
 - `host_shared(list)`
 - `device(expression)`

3-119

OpenMP Accelerator Subcommittee

- ❑ `!$omp acc_data [clause]...`
`#pragma omp acc_data [clause]...`
 - `acc_shared(list)`
 - `acc_copy(list)`
 - `acc_copyin(list)`
 - `acc_copyout(list)`
 - `present(list)`
 - `host_shared(list)`
 - `device(expression)`

3-120

OpenMP Accelerator Subcommittee

- `!$omp acc_loop [clause]...`
`#pragma omp acc_loop [clause]...`
 - `cache(obj:depth)`
 - `collapse(n)`
 - `level(n)`
 - `num_pes(depth:n)`
 - `reduction(operator:name)`
 - `schedule(sched-name)`
 - `hetero(n, width)`

3-121

The Portland Group®

PGI's Dilemma

- Use a standard, but one that fits badly
 - some programs will run well, most will not
 - does not take advantage of accelerator strengths
- OR, define a new model that fits well
 - multidimensional parallelism
 - multivariate scheduling options
 - defined data movement points
 - use compiler feedback to drive porting / tuning

3-122

The Portland Group®

Availability and Additional Information

- PGI Accelerator Programming Model – **is supported for x64+NVIDIA targets in the PGI Fortran and C compilers, available now**
- Other GPU and Accelerator Targets – **are being studied by PGI, and may be supported in the future as the necessary low-level software infrastructure (e.g. OpenCL) becomes more widely available**
- Further Information – see www.pgroup.com/accelerate for a detailed specification of the PGI Accelerator model, an FAQ, and related articles and white papers

3-123

The Portland Group®

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>

The Portland Group®

Copyright Notice

© Contents copyright 2009-2011, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.

3-125

