Programming GPUs with OpenACC
Part 3: Advanced Topics

Michael Wolfe

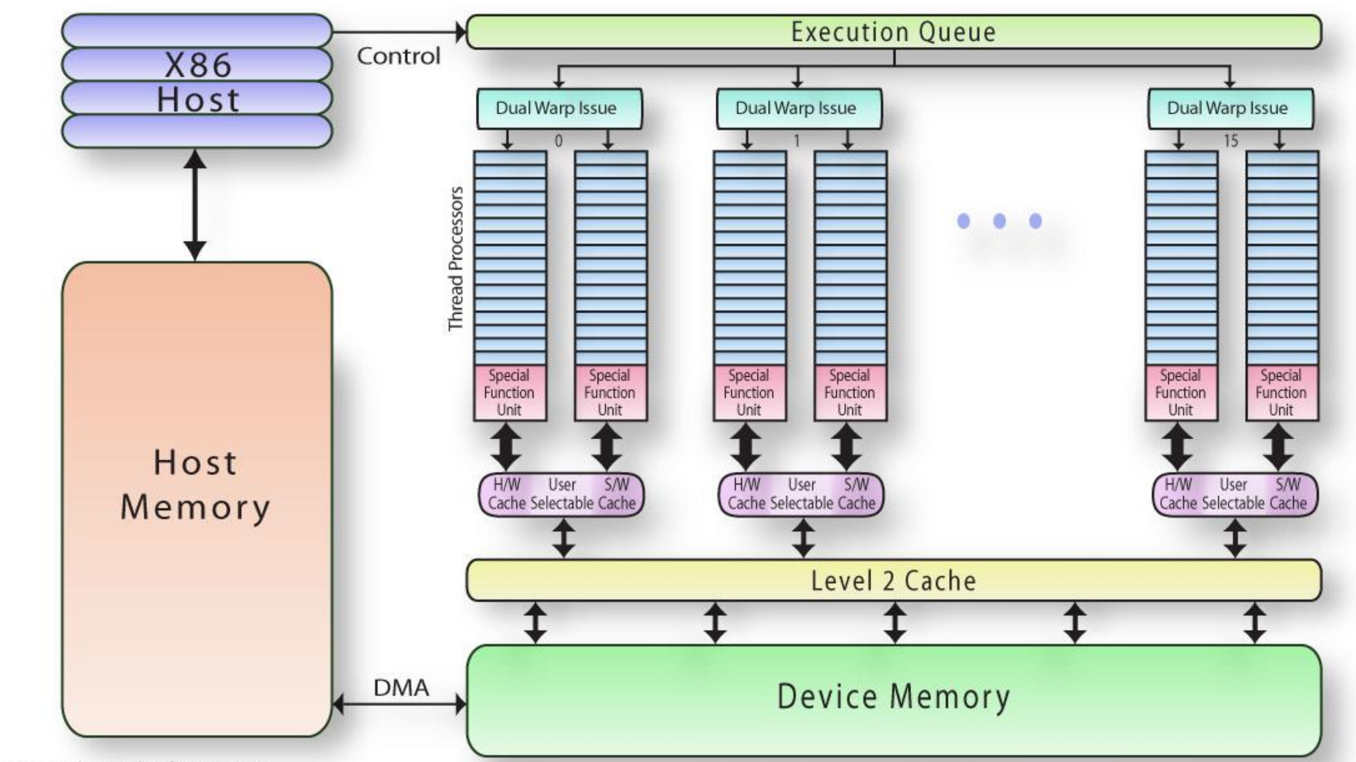
Michael.Wolfe@pgroup.com

http://www.pgroup.com

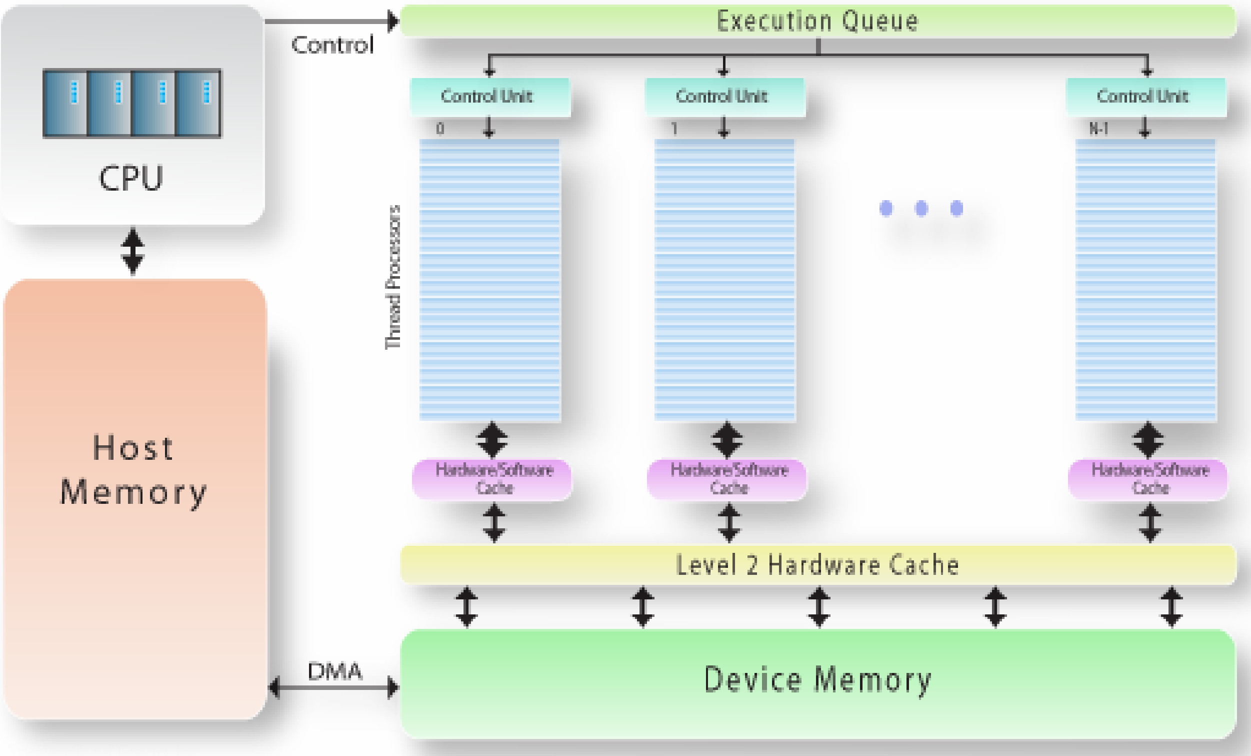
May 2012

Questions

Abstracted x64+Fermi Accelerator Architecture



Abstract Target



Basic Topics

- **Manage data movement between host and GPU**
- **Lots of parallelism (think nested parallel loops)**
- **Data layout matters (stride-1 in the parallel loop)**
- **The parallelism *schedule* matters**

Advanced Topics

- Asynchronous Data Movement and Execution
- Data Caching (shared memory)
- Parallel and Kernels constructs
- Data regions and procedures, Present clause, Update directive
- Multiple GPUs, OpenMP and MPI
- Splitting work between GPU and host
- Mixing with CUDA (C and Fortran)
- Procedure calls, inlining
- Optimizing kernels
- C-specific features and issues
- PGI-specific features and issues
- Future issues

Data Region across Procedures

```
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];

}
```

Data Region across Procedures

```
subroutine domany(...)


  call saxpy( n, a, x, y )
```

```
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

```
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];

}
```

Data Region across Procedures

```
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
```

Data Region across Procedures

```
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])\
    present_or_copy(y[0:n])
 for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
```

Data Region across Procedures

```
subroutine domany(...)

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

  call saxpy( n, a, x2, y2 )
```

```
subroutine saxpy( n, a, x, y )
 integer :: n
 real :: a, x(*), y(*)
 integer :: i
 !$acc kernels loop &
    present_or_copyin(x(1:n)) &
    present_or_copy(y(1:n))
 do i = 1, n
  y(i) = y(i) + a*x(i)
 enddo
end subroutine
```

Data Region across Procedures

```
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
 for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
```

Data Region across Procedures

```
subroutine domany(...)

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

  call saxpy( n, a, x2, y2 )
```

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

Data Region across Procedures

```
subroutine domany(...)

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

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

Update

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


  MPI_SENDRECV( x, ... )



   ...adjust...
}
```

```
#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

```
#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
...
   }
}
```

Async

- synchronous – directive / construct does not complete until action is complete
- asynchronous – program will continue beyond directive / construct before action is complete

Async

```
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];

}
```

Async

```
subroutine domany(...)

!$acc data copy( y(:,:), x(:) )
  do j = 1, m
   call saxpy(n,a,x,y(:,j),j)
  enddo
  !$acc wait  ! waits for all
!$acc end data
```

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

Host + GPU

```
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
```

Host + GPU

```
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

```
#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

```
#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

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

Procedure calls on the device

```
#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>`

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		

PGI Fortran Intrinsics

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	

other functions

- PGI libm routines
  - `use libm`
  - `#include <accelmath.h>`
- PGI device builtin routines
  - `use cudadevice`
  - `#include <cudadevice.h>`

Parallel vs Kernels

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

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

}
```

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

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

}
```

Parallel vs. Kernels

```
!$acc kernels loop
 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
```

Parallel vs. Kernels

```
!$acc kernels loop gang, vector(8)
 do j = js, je
  !$acc loop gang, vector(32)
  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
```

Parallel vs. Kernels

```
!$acc kernels loop gang, worker(8)
 do j = js, je
  !$acc loop vector(32)
  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
```

Parallel vs. Kernels

```
!$acc parallel loop gang, worker
 do j = js, je
  !$acc loop vector
  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
```

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]);
}
```

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 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
```

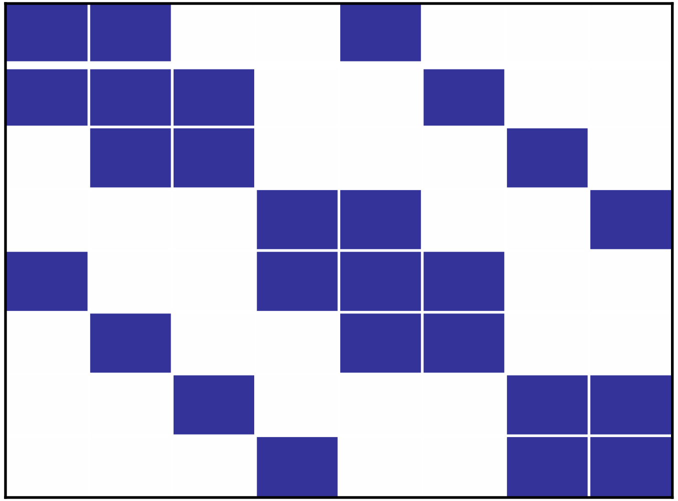
Mixing OpenACC with CUDA Fortran (PGI)

```
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(...)
  !$acc parallel
   do i = 1, n
     call devproc(a(i))
   enddo
  !$acc end parallel
 end subroutine
end module
```

Diagonal Representation Sparse Matrix



7	2			5			
9	6	4			7		
	8	2	0			9	
		0	7	8			6
1			7	5	4		
	3			3	3	0	
		4			0	4	2
			1			1	3

Diagonal Representation Sparse Matrix

		7	2	5
	9	6	4	7
	8	2	0	9
	0	7	8	6
1	7	5	4	
3	3	3	0	
4	0	4	2	
1	1	3		

-4	-1	0	1	4

7	2			5			
9	6	4			7		
	8	2	0			9	
		0	7	8			6
1			7	5	4		
	3			3	3	0	
		4			0	4	2
			1			1	3

Sparse Matrix-Vector Multiply

		7	2	5
	9	6	4	7
	8	2	0	9
	0	7	8	6
1	7	5	4	
3	3	3	0	
4	0	4	2	
1	1	3		

-4	-1	0	1	4

```
for( i = 0; i < nrows; ++i ){
   float val = 0.0f;
   for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
       val += data[i*nzeros+d] * x[j];
   }
   y[i] = val;
}
```

Sparse Matrix-Vector Multiply

```
for( i = 0; i < nrows; ++i ){
  float val = 0.0f;
  for( d = 0; d < nzeros; ++d ){
    j = i + offset[d];
    if( j >= 0 && j < nrows )
      val += m[i*nzeros+d] * v[j];
  }
  x[i] = val;
}
```

Sparse Matrix-Vector Multiply

```
#pragma acc parallel loop copyin( m[0:nzeros*nrows], v[0:nrows] )
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i*nzeros+d] * v[j];
    }
    x[i] = val;
  }
```

Sparse Matrix-Vector Multiply

```
#pragma acc parallel loop copyin( m[0:nzeros*nrows], v[0:nrows] )
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i+nrows*d] * v[j];
    }
    x[i] = val;
  }
```

Sparse Matrix-Vector Multiply

```
#pragma acc parallel loop deviceptr( m, v, offset, x )
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i+nrows*d] * v[j];
    }
    x[i] = val;
  }
```

C-specific Features and Issues

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

```
a*1.0    vs.  a*1.0f
sin(a)   vs.  sin(a)
```

- Pointer disambiguation matters

```
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

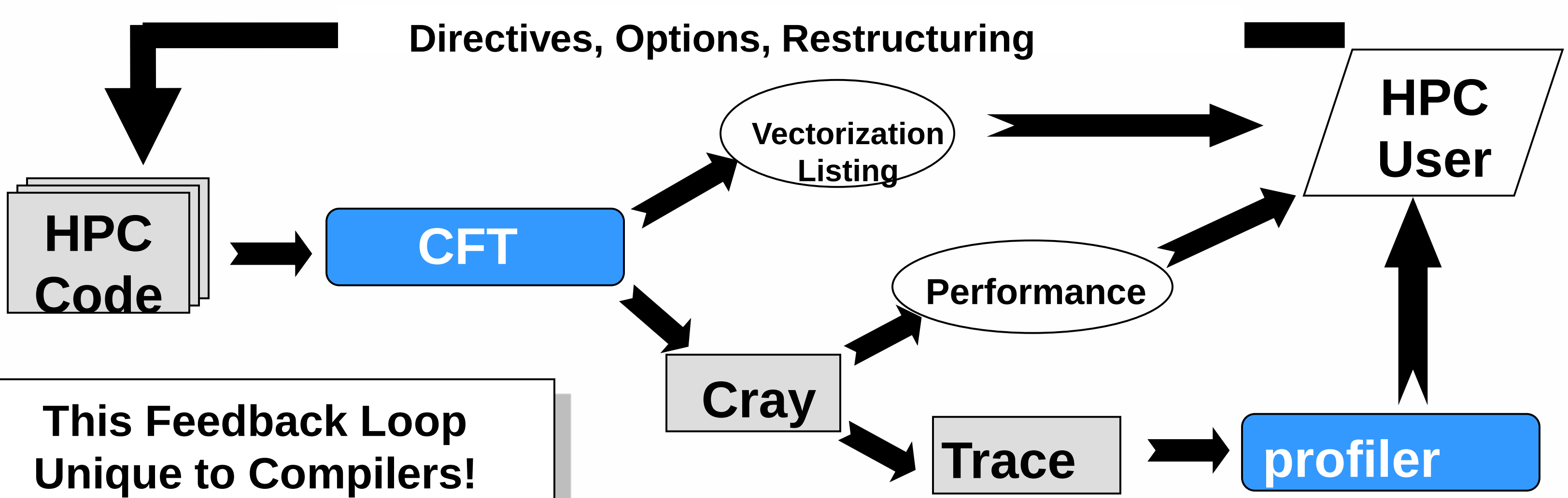
PGI new functions

```
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 );
```

How did we make Vectors Work?

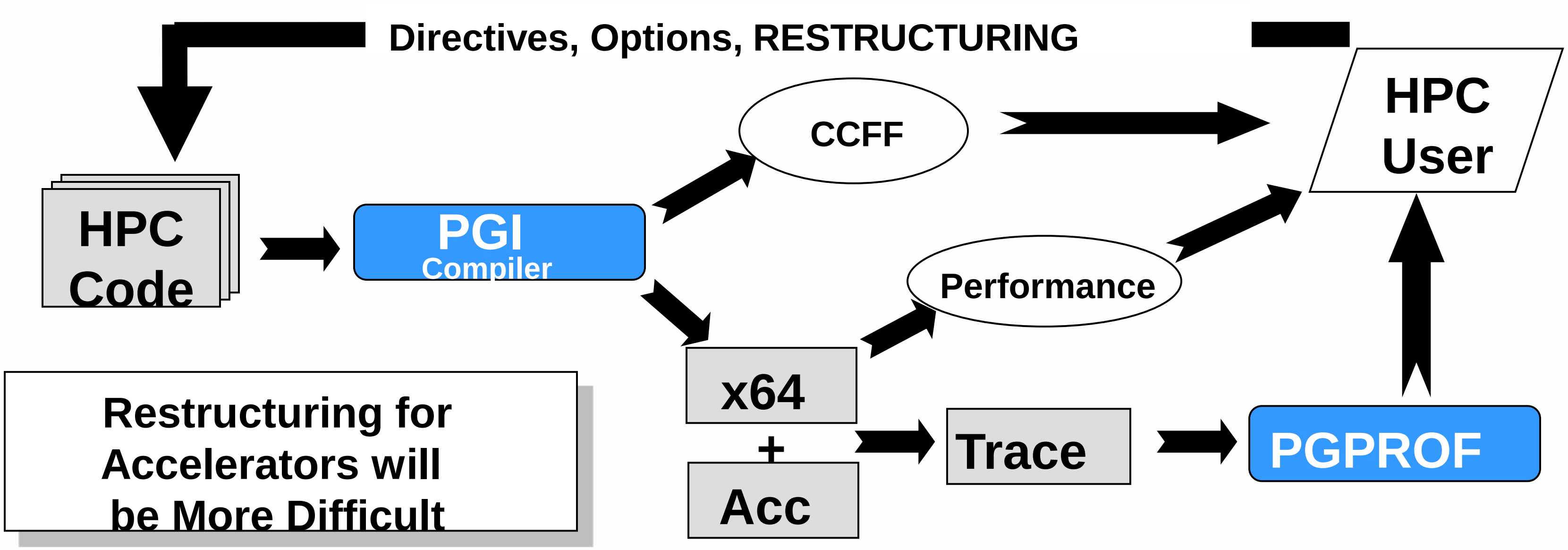
Compiler-to-Programmer Feedback – a classic "Virtuous Cycle"



This Feedback Loop
Unique to Compilers!

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

Compiler-to-Programmer Feedback



Compiler-to-User Feedback

```
% pgfortran -fast -acc -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, !$acc loop gang, vector(16)
          8, !$acc loop gang, vector(16)
```

Async on Data construct

```
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
- -ta=nvidia sets target accelerator (default)
- -ta=nvidia,cc1x cc10 cc11 cc12 cc13 cc2x cc20 [cc3x cc30]
  - sets compute capability(ies)
- -ta=nvidia,fastmath uses fast math versions (less accurate)
- -ta=nvidia,cuda4.0 cuda4.1 [cuda4.2] sets toolkit version
- -ta=nvidia,nofma avoids use of fused mul-add (precision diffs)
- -ta=nvidia,O0 O1 O2 O3 sets device code opt level
- -ta=nvidia,keepgpu lets you look at generate GPU code

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

Where to get help

- OpenACC Forum – www.openacc.org/forum
- OpenACC documentation – www.openacc.org/downloads
- PGI Licensed Customer Support - trs@pgroup.com
- PGI User's Forum – www.pgroup.com/userforum/index.php
- PGI Articles – www.pgroup.com/resources/articles.htm
  www.pgroup.com/resources/accel.htm
- PGI User's Guide – www.pgroup.com/doc/pgiug.pdf
- CUDA Fortran Reference Guide – www.pgroup.com/doc/pgicudafortug.pdf

Copyright Notice