GPU Computing with OpenACC Directives
GPUs Reaching Broader Set of Developers

- Universities
- Supercomputing Centers
- Oil & Gas
- CAE
- CFD
- Finance
- Rendering
- Data Analytics
- Life Sciences
- Defense
- Weather
- Climate
- Plasma Physics

Early Adopters

100,000's

1,000,000's

2004

Present

Time
3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
OpenACC Directives

Program myscience

... serial code ...

!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo

!$acc end kernels
...
End Program myscience

CPU

GPU

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
main() {
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
OpenACC
The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU
Directives: Easy & Powerful

Real-Time Object Detection
Global Manufacturer of Navigation Systems

Valuation of Stock Portfolios using Monte Carlo
Global Technology Consulting Company

Interaction of Solvents and Biomolecules
University of Texas at San Antonio

5x in 40 Hours
2x in 4 Hours
5x in 8 Hours

“Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.”

-- Developer at the Global Manufacturer of Navigation Systems
Focus on Exposing Parallelism

With Directives, tuning work focuses on exposing parallelism, which makes codes inherently better.

Example: Application tuning work using directives for new Titan system at ORNL

**S3D**
Research more efficient combustion with next-generation fuels

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

**CAM-SE**
Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
  

- Quick reference card also available

- Beta implementations available now from PGI, Cray, and CAPS
A Very Simple Exercise: SAXPY

**SAXPY in C**

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

**SAXPY in Fortran**

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

...$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...```
Directive Syntax

- **Fortran**
  ```fortran
  !$acc directive [clause [,] clause] ...
  ```
  Often paired with a matching end directive surrounding a structured code block
  ```fortran
  !$acc end directive
  ```

- **C**
  ```c
  #pragma acc directive [clause [,] clause] ...
  ```
  Often followed by a structured code block
Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do

  do i=1,n
    a(i) = b(i) + c(i)
  end do

!$acc end kernels
```

**Kernel:**
A parallel function that runs on the GPU
Kernels Construct

Fortran

```fortran
!$acc kernels [clause ...]  
  structured block  
!$acc end kernels
```

Clauses

```
if( condition )  
async( expression )
```

Also, any data clause (more later)

C

```c
#pragma acc kernels [clause ...]  
  { structured block }
```
C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.
    ```c
    float *restrict ptr
    ```
  - Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*

- Limits the effects of pointer aliasing

- OpenACC compilers often require `restrict` to determine independence
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

Trivial first example

Apply a loop directive

Learn compiler commands

```c
#include <stdlib.h>

void saxpy(int n, float a, float *x, float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

*restrict: “I promise y does not alias x”*
Compile and run

C:
```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

Fortran:
```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
```

Compiler output:
```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
  Generating copy(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
  CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
  CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```
Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: \( \nabla^2 f(x,y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
while ( error > tol && iter < iter_max ) {
    error=0.0;
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Jacobi Iteration Fortran Code

do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
      A(i, j-1) + A(i, j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  iter = iter +1
end do
while ( error > tol && iter < iter_max ) {
    error=0.0;
    
    #pragma omp parallel for shared(m, n, Anew, A)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    
    #pragma omp parallel for shared(m, n, Anew, A)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
OpenMP Fortran Code

```fortran
!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
    do j=1,m
        do i=1,n
            Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                    A(i  , j-1) + A(i  , j+1))
            err = max(err, Anew(i,j) - A(i,j))
        end do
    end do
!$omp parallel do shared(m,n,Anew,A)
    do j=1,m-2
        do i=1,n-2
            A(i,j) = Anew(i,j)
        end do
    end do
iter = iter + 1
end do
```

Parallelize loop across CPU threads
Access to BigRed2

- ssh <username>@bigred2.uits.iu.edu
- cp -r /N/u/jbentz/BigRed2/oct2/openacc .
- cd openacc
- module swap PrgEnv-cray PrgEnv-pgi

Use batch system for job submission
- qsub—submit a job to the queue
- qstat—show all jobs in the queue
- qdel—delete a job from the queue
Exercises: General Instructions (compiling)

Exercises are in “openacc/exercises” directory
  Solutions are in “openacc/exercise_solutions” directory

To compile, use one of the provided makefiles
  C:
    > make
  Fortran:
    > make -f Makefile_f90

Remember these compiler flags:
  -acc -ta=nvidia -Minfo=accel
Exercises: General Instructions (running)

To run, just execute on the command line
OpenACC looks like this

```
> qsub runit.acc
```

The OpenMP version specifies number of cores to use

```
> qsub runit.omp
```
Exercise 1: Jacobi Kernels

Task: use acc kernels to parallelize the Jacobi loop nests

Edit laplace2D.c or laplace2D.f90 (your choice)
- In the 001-laplace2D-kernels directory
- Add directives where it helps
- Figure out the proper compilation command (similar to SAXPY example)
  - Hint: -acc -ta=nvidia -Minfo=accel
- Compile both with and without OpenACC parallelization
- Optionally compile with OpenMP (original code has OpenMP directives)
- Run OpenACC version with laplace_acc, OpenMP with laplace_omp

Q: can you get a speedup with just kernels directives?
- Versus 1 CPU core? Versus 8 CPU cores?
while ( error > tol && iter < iter_max ) { 
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + 
                                 A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                 A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

!$acc kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels

iter = iter +1
end do
Exercise 1 Solution: C Makefile

CC       = pgcc
CCFLAGS  =
ACCFLAGS = -acc -ta=nvidia -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN = laplace2d_omp laplace2d_acc

all: $(BIN)

laplace2d_acc: laplace2d.c
  $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_omp: laplace2d.c
  $(CC) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
  $(RM) $(BIN)
Exercise 1 Solution: Fortran Makefile

F90 = pgf90
CCFLAGS =
ACCFLAGS = -acc -ta=nvidia -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN = laplace2d_f90_omp laplace2d_f90_acc

all: $(BIN)

laplace2d_f90_acc: laplace2d.f90
  $(F90) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_f90_omp: laplace2d.f90
  $(F90) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
  $(RM) $(BIN)
Exercise 1: Compiler output (C)

```
pgcc -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
  56, Generating present_or_copyin(A[0:][0:]
  Generating present_or_copyout(Anew[1:4094][1:4094])
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  57, Loop is parallelizable
  59, Loop is parallelizable
     Accelerator kernel generated
     57, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
     59, #pragma acc loop gang, vector(64) /* blockIdx.x threadIdx.x */
     CC 1.3 : 20 registers; 48 shared, 32 constant, 0 local memory bytes
     CC 2.0 : 26 registers; 0 shared, 72 constant, 0 local memory bytes
  63, Max reduction generated for error
  68, Generating present_or_copyout(A[1:4094][1:4094])
  Generating present_or_copyin(Anew[1:4094][1:4094])
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  69, Loop is parallelizable
  71, Loop is parallelizable
     Accelerator kernel generated
     69, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
     71, #pragma acc loop gang, vector(64) /* blockIdx.x threadIdx.x */
     CC 1.3 : 16 registers; 40 shared, 8 constant, 0 local memory bytes
     CC 2.0 : 19 registers; 0 shared, 56 constant, 0 local memory bytes
```
Exercise 1: Performance

CPU: AMD Opteron 6276
16 Cores @ 2.3 GHz

GPU: NVIDIA Tesla K20

<table>
<thead>
<tr>
<th>Execution (4096x4096)</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>99.4</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>87.5</td>
<td>1.13x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>61.1</td>
<td>1.62x</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>53.7</td>
<td>1.85x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>185.7</td>
<td>0.29x FAIL</td>
</tr>
</tbody>
</table>

Speedup vs. 1 CPU core
Speedup vs. 6 CPU cores
What went wrong?

Add `-ta=nvidia,time` to compiler command line

```
/N/hd00/jbentz/BigRed2/oct2/openacc_test/exercise_solutions/001-laplace2D-kernels/laplace2d.c
```

```
main
  68: region entered 1000 times
time(us): total=110,608,231 init=298 region=110,607,933
kernels=1,887,589 data=108,220,077
w/o init: total=110,607,933 max=135,978 min=109,965 avg=110,607
  71: kernel launched 1000 times
grid: [64x1024] block: [64x4]
time(us): total=1,887,589 max=1,895 min=1,881 avg=1,887
/N/hd00/jbentz/BigRed2/oct2/openacc_test/exercise_solutions/001-laplace2D-kernels/laplace2d.c
```

```
main
  56: region entered 1000 times
time(us): total=76,001,686 init=512,975 region=75,488,711
kernels=3,976,231 data=68,863,445
w/o init: total=75,488,711 max=129,045 min=74,659 avg=75,488
  59: kernel launched 1000 times
grid: [64x1024] block: [64x4]
time(us): total=3,834,516 max=5,184 min=3,828 avg=3,834
  63: kernel launched 1000 times
grid: [1] block: [256]
time(us): total=141,715 max=1,518 min=139 avg=141
```

```
total: 186.612793 s
```

Huge Data Transfer Bottleneck!
Computation: 5.8 seconds
Data movement: 177 seconds
Basic Concepts

For efficiency, decouple data movement and compute off-load.
**Excessive Data Transfers**

```c
while ( error > tol && iter < iter_max ) {
    error=0.0;
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
}

*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!*
```
Data Construct

**Fortran**

```fortran
!$acc data [clause ...]
  structured block
!$acc end data
```

**C**

```c
#pragma acc data [clause ...]
  { structured block }
```

**General Clauses**

```c
if( condition )
  async( expression )
```

Manage data movement. Data regions may be nested.
**Data Clauses**

- **copy (list)** Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

- **copyin (list)** Allocates memory on GPU and copies data from host to GPU when entering region.

- **copyout (list)** Allocates memory on GPU and copies data to the host when exiting region.

- **create (list)** Allocates memory on GPU but does not copy.

- **present (list)** Data is already present on GPU from another containing data region.

And **present_or_copy[in|out]**, **present_or_create**, **deviceptr**.
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- C

```c
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```

- Fortran

```fortran
!$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```

- Note: data clauses can be used on data, kernels or parallel
Update Construct

Fortran

```fortran
!$acc update [clause ...]
```

Clauses

```fortran
host( list )
device( list )
```

C

```c
#pragma acc update [clause ...]
```

```c
if( expression )
async( expression )
```

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.
Data movement can be conditional, and asynchronous.
Exercise 2: Jacobi Data Directives

Task: use acc data to minimize transfers in the Jacobi example.

Start from given laplace2D.c or laplace2D.f90 (your choice)
- In the 002-laplace2d-data directory
- Add directives where it helps (hint: [do] while loop)

Q: What speedup can you get with data + kernels directives?
- Versus 1 CPU core? Versus 8 CPU cores?
Exercise 2 Solution: OpenACC C

```c
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator.
Exercise 2 Solution: OpenACC Fortran

```fortran
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
   err=0._fp_kind

!$acc kernels
   do j=1,m
      do i=1,n

         Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                     A(i  , j-1) + A(i  , j+1))

         err = max(err, Anew(i,j) - A(i,j))
      end do
   end do
!$acc end kernels

... ...

iter = iter +1
end do
!$acc end data
```
## Exercise 2: Performance

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>53.7</td>
<td>--</td>
</tr>
<tr>
<td>OpenACC GPU K20</td>
<td>185.7</td>
<td>0.29x</td>
</tr>
<tr>
<td>OpenACC GPU K20-opt</td>
<td>6.6</td>
<td>8.1x</td>
</tr>
</tbody>
</table>

*Speedups vs. 8 CPU cores*
Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code

- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance

- Will tackle these in later exercises
Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.
Tips and Tricks

- (PGI) Use time option to learn where time is being spent
  - `-ta=nvidia, time`
- Eliminate pointer arithmetic
- Inline function calls in directives regions
  - (PGI): `-inline` or `-inline, levels(<N>)`
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro
OpenACC Learning Resources

OpenACC info, specification, FAQ, samples, and more
  http://openacc.org

PGI OpenACC resources
  http://www.pgroup.com/resources/accel.htm
Thank you!