Introduction to OpenACC

Jeff Larkin, NVIDIA
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility
OpenACC
The Standard for GPU Directives

- **Simple:** 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
High-level

- Compiler directives to specify parallel regions in C & Fortran
  - Offload parallel regions
  - Portable across OSes, host CPUs, accelerators, and compilers

- Create high-level heterogeneous programs
  - Without explicit accelerator initialization
  - Without explicit data or program transfers between host and accelerator
High-level... with low-level access

- Programming model allows programmers to start simple
- Compiler gives additional guidance
  - Loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.
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
  www.openacc.org
- Quick reference card also available
- Compilers available now from PGI, Cray, and CAPS
Exposing Parallelism with OpenACC
A Very Simple Exercise: SAXPY

**SAXPY in C**

```c
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
    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(n), y(n), a
    integer :: n, i
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
end subroutine saxpy
```

... ! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
A Very Simple Exercise: SAXPY OpenMP

**SAXPY in C**

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma omp parallel for
    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(n), y(n), a
    integer :: n, i

    !$omp parallel do
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$omp end parallel do
end subroutine saxpy

... ! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
A Very Simple Exercise: SAXPY OpenACC

**SAXPY in C**

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc parallel loop
    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(n), y(n), a
    integer :: n, i

    !$acc parallel loop
do i=1,n
    y(i) = a*x(i)+y(i)
enddo
 !$acc end parallel loop
end subroutine saxpy
...!

... // Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d);
...```
OpenACC is not **GPU Programming**.

OpenACC is **Exposing Parallelism** in your code.
OpenACC Execution Model

GPU

Generate Parallel Code for GPU

Compute-Intensive Functions

$\texttt{acc parallel}$

Application Code

$\texttt{acc end parallel}$

Rest of Sequential CPU Code

CPU
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

- **Common Clauses**

  - `if(condition)`
  - `async(handle)`
OpenACC parallel Directive

Programmer identifies a loop as having parallelism, compiler generates a parallel kernel for that loop.

```c
!acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!acc end parallel loop
```

*Most often parallel will be used as parallel loop.*

Kernel:
A parallel function that runs on the GPU
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 parallel loop
    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;
}
```
Compile (PGI)

C:

```
pgcc -acc [-Minfo=accel] [-ta=nvidia] -o saxpy_acc saxpy.c
```

Fortran:

```
pgf90 -acc [-Minfo=accel] [-ta=nvidia] -o saxpy_acc saxpy.f90
```

Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  11, Accelerator kernel generated
  13, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
  11, Generating present_or_copyin(x[0:n])
  Generating present_or_copy(y[0:n])
  Generating NVIDIA code
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  Generating compute capability 3.0 binary
```
The PGI compiler provides automatic instrumentation when `PGI_ACC_TIME=1` at runtime.

Accelerator Kernel Timing data
/home/jlarkin/kernels/saxpy/saxpy.c

  saxpy  NVIDIA  devicenum=0
  time(us):  3,256
  11: data copyin reached 2 times
      device time(us): total=1,619 max=892 min=727 avg=809
  11: kernel launched 1 times
      grid: [4096] block: [256]
      device time(us): total=714 max=714 min=714 avg=714
      elapsed time(us): total=724 max=724 min=724 avg=724
  15: data copyout reached 1 times
      device time(us): total=923 max=923 min=923 avg=923
Run

The Cray compiler provides automatic instrumentation when `CRAY_ACC_DEBUG=<1,2,3>` at runtime

ACC: Initialize CUDA
ACC: Get Device 0
ACC: Create Context
ACC: Set Thread Context
ACC: Start transfer 2 items from saxpy.c:17
   allocate, copy to acc 'x' (4194304 bytes)
   allocate, copy to acc 'y' (4194304 bytes)
ACC: End transfer (to acc 8388608 bytes, to host 0 bytes)
ACC: Execute kernel saxpy$ck_L17_1 blocks:8192 threads:128 async(auto) from saxpy.c:17
ACC: Wait async(auto) from saxpy.c:18
ACC: Start transfer 2 items from saxpy.c:18
   free 'x' (4194304 bytes)
   copy to host, free 'y' (4194304 bytes)
ACC: End transfer (to acc 0 bytes, to host 4194304 bytes)
Another approach: kernels construct

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```plaintext
!$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
```

The compiler identifies 2 parallel loops and generates 2 kernels.
OpenACC parallel vs. kernels

<table>
<thead>
<tr>
<th>PARALLEL</th>
<th>KERNELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Requires analysis by</td>
<td>• Compiler performs parallel analysis and parallelizes what</td>
</tr>
<tr>
<td>programmer to ensure safe</td>
<td>it believes safe</td>
</tr>
<tr>
<td>parallelism</td>
<td>• Can cover larger area of code with single directive</td>
</tr>
<tr>
<td>• Straightforward path from</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td></td>
</tr>
</tbody>
</table>

Both approaches are equally valid and can perform equally well.
OpenACC by Example
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 ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            err = max(err, 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: OpenMP C Code

```c
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
        }
    }
    err = max(err, 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++;
}
```
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
    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]);

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

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

    iter++;
}
PGI Accelerator Compiler output (C)

pgcc -Minfo=all -ta=nvidia:5.0,cc3x -acc -Minfo=accel -o laplace2d_acc laplace2d.c

main:

56, Accelerator kernel generated
57, #pragma acc loop gang /* blockIdx.x */
59, #pragma acc loop vector(256) /* threadIdx.x */

56, Generating present_or_copyin(A[0:][0:])
Generating present_or_copyout(Anew[1:4094][1:4094])
Generating NVIDIA code
Generating compute capability 3.0 binary

59, Loop is parallelizable

68, Accelerator kernel generated
69, #pragma acc loop gang /* blockIdx.x */
71, #pragma acc loop vector(256) /* threadIdx.x */

68, Generating present_or_copyout(A[1:4094][1:4094])
Generating present_or_copyin(Anew[1:4094][1:4094])
Generating NVIDIA code
Generating compute capability 3.0 binary

71, Loop is parallelizable
### Performance

**CPU:** AMD IL-16 @ 2.2 GHz  
**GPU:** NVIDIA Tesla K20X

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>109.7</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>71.6</td>
<td>1.5x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>53.7</td>
<td>2.0x</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>65.5</td>
<td>1.7x</td>
</tr>
<tr>
<td>CPU 16 OpenMP threads</td>
<td>66.7</td>
<td>1.6x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>180.9</td>
<td>0.6x FAIL!</td>
</tr>
</tbody>
</table>
What went wrong?

Set PGI_ACC_TIME environment variable to ‘1’

Accelerator Kernel Timing data
/lustre/scratch/jlarkin/openacc-workshop/exercises/001-laplace2D-kernels/laplace2d.c

main

69: region entered 1000 times
time(us): total=109,998,808 init=262 region=109,036,546
kernels=1,748,221 data=109,554,793
w/o init: total=109,998,546 max=110,762 min=109,378 avg=109,501
69: kernel launched 1000 times
grid: [4094] block: [256]
time(us): total=1,748,221 max=1,820 min=1,727 avg=1,748

/lustre/scratch/jlarkin/openacc-workshop/exercises/001-laplace2D-kernels/laplace2d.c
main

57: region entered 1000 times
time(us): total=71,790,531 init=491,553 region=71,298,978
kernels=2,369,807 data=68,968,929
w/o init: total=71,298,978 max=75,603 min=70,486 avg=71,298
57: kernel launched 1000 times
grid: [4094] block: [256]
time(us): total=2,347,795 max=3,737 min=2,343 avg=2,347

58: kernel launched 1000 times
grid: [1] block: [256]
time(us): total=22,012 max=1,400 min=19 avg=22

total: 181.792123 s

109.5 seconds

1.7 seconds

Huge Data Transfer Bottleneck!

Computation: 4.1 seconds
Data movement: 178.4 seconds

68.9 seconds

2.4 seconds
Basic Concepts

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

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

And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!
Data Management with OpenACC
Defining data regions

The data construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

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

Data Region

do i=1,n
   a(i) = b(i) + c(i)
end do
!$acc end data
```

Arrays a, b, and c will remain on the GPU until the end of the data region.
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
  ```
  #pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
  ```

- Fortran
  ```
  !$acc data copyin(a(1:end)), copyout(b(s/4:3*s/4))
  ```

- Note: data clauses can be used on data, parallel, or kernels
Task: use acc data to minimize transfers in the Jacobi example
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
        for( int j = 1; j < n-1; j++) {
            for(int i = 1; i < m-1; i++) {


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

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

    iter++;
}
Did it help?

Accelerator Kernel Timing data
/lustre/scratch/jlarkin/openacc-workshop/exercises/001-laplace2D-kernels/laplace2d.c

main
69: region entered 1000 times
  time(us): total=1,791,050 init=217 region=1,790,833
  kernels=1,742,066
w/o init: total=1,790,833 max=1,950 min=1,773 avg=1,790
69: kernel launched 1000 times
  grid: [4094]  block: [256]
  time(us): total=1,742,066 max=1,809 min=1,725 avg=1,742

main
57: region entered 1000 times
  time(us): total=2,710,902 init=182 region=2,710,720
  kernels=2,361,193
w/o init: total=2,710,720 max=4,163 min=2,697 avg=2,710
57: kernel launched 1000 times
  grid: [4094]  block: [256]
  time(us): total=2,339,800 max=3,709 min=2,334 avg=2,339
58: kernel launched 1000 times
  grid: [1]  block: [256]
  time(us): total=21,393 max=1,321 min=19 avg=21

main
51: region entered 1 time
  time(us): total=5,063,688 init=489,133 region=4,574,555
  data=68,993
w/o init: total=4,574,555 max=4,574,555 min=4,574,555 avg=4,574,555

0.69 seconds
### Performance

**CPU:** AMD IL-16 @ 2.2 GHz  
**GPU:** NVIDIA Tesla K20X

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>109.7</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>71.6</td>
<td>1.5x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>53.7</td>
<td>2.0x</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>65.5</td>
<td>1.7x</td>
</tr>
<tr>
<td>CPU 16 OpenMP threads</td>
<td>66.7</td>
<td>1.6x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>4.96</td>
<td>10.8x</td>
</tr>
</tbody>
</table>
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

- More on this in the Advanced OpenACC session this afternoon.
OpenACC Tips & Tricks
C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.
  
  ```
  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

- Compilers often require restrict to determine independence (true for OpenACC, OpenMP, and vectorization)
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined

Tips and Tricks

- Nested loops are best for parallelization
  - Large loop counts (1000s) needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: use restrict keyword in C
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Inline function calls in directives regions
  - (PGI): -Minline or -Minline=levels:<N>
  - (Cray): -hpl=<dir/>
  - This has been improved in OpenACC 2.0
Tips and Tricks (cont.)

- Use time option to learn where time is being spent
  - (PGI) $\text{PGI\_ACC\_TIME}=1$ (runtime environment variable)
  - (Cray) $\text{CRAY\_ACC\_DEBUG}=<1,2,3>$ (runtime environment variable)
  - (CAPS) $\text{HMPPRT\_LOG\_LEVEL}=\text{info}$ (runtime environment variable)

- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.

- Use contiguous memory for multi-dimensional arrays

- Use data regions to avoid excessive memory transfers

- Conditional compilation with \_\_OPENACC macro
Thank you