Getting Started with OpenACC
Jeff Larkin, NVIDIA
GPUs Reaching Broader Set of Developers

1,000,000's

100,000's

Early Adopters

2004

Present

Time

CAE
CFD
Finance
Rendering
Data Analytics
Life Sciences
Defense
Weather
Climate
Plasma Physics

Research

Universities
Supercomputing Centers
Oil & Gas

100,000's

2004

Present

Time

1,000,000's

Research

Universities
Supercomputing Centers
Oil & Gas

100,000's

2004

Present

Time

1,000,000's

Research

Universities
Supercomputing Centers
Oil & Gas

Early Adopters

2004

Present

Time

1,000,000's

Research

Universities
Supercomputing Centers
Oil & Gas

Early Adopters

2004

Present

Time

1,000,000's

Research

Universities
Supercomputing Centers
Oil & Gas

Early Adopters
3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
OpenACC
Open Programming Standard for Parallel Computing

“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

“OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”

--Michael Wong, CEO OpenMP Directives Board

OpenACC Standard
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 Expressing Parallelism

With Directives, tuning work focuses on expressing 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 is not GPU Programming.

OpenACC is Expressing Parallelism in your code.
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
- Public Comment Draft of 2.0 Specification now available online.

  [www.openacc.org](http://www.openacc.org)

- Quick reference card also available
- Compilers available now from PGI, Cray, and CAPS
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator
Tools for quick ramp

www.nvidia.com/gpudirectives
Expressing 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 Execution Model

1. Generate parallel code for GPU
2. Allocate GPU memory and copy input data
3. Execute parallel code on GPU
4. Copy output data to CPU and deallocate GPU memory

\[
\text{Compute-Intensive Code}
\]

\[
\text{Rest of Sequential CPU Code}
\]
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**

```c
if(condition), async(handle)
```
OpenACC parallel Directive

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

```fortran
%!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 function that runs in parallel on the GPU
Complete SAXPY example code

- 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

```plaintext
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
  ACC: allocate, copy to acc 'x' (4194304 bytes)
  ACC: 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
  ACC: free 'x' (4194304 bytes)
  ACC: 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.

```c
!$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

**PARALLEL**
- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP

**KERNELS**
- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with single directive

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: $\nabla^2 f(x, y) = 0$

\[
A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}
\]
Jacobi Iteration: C Code

```c
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++;
}
```

Iterate until converged
Iterate across matrix elements
Calculate new value from neighbors
Compute max error for convergence
Swap input/output arrays
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++) {

            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 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++;
}
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
Execution Time (lower is better)

CPU: Intel i7-3930K
6 Cores @ 3.20GHz

GPU: NVIDIA Tesla K20

OpenACC code is SLOWER than even serial code. Why?
What went wrong?

Set `PGI_ACC_TIME` environment variable to ‘1’

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

```
main NVIDIA devicenum=0
time(us): 93,201,190
56: data copyin reached 1000 times
device time(us): total=23,049,452 max=28,928 min=22,761 avg=23,049
56: kernel launched 1000 times
grid: [4094] block: [256]
device time(us): total=2,609,928 max=2,812 min=2,593
elapsed time(us): total=2,872,585 max=3,022 min=2,642
56: reduction kernel launched 1000 times
grid: [1] block: [256]
device time(us): total=19,218 max=724 min=16 avg=19
elapsed time(us): total=29,070 max=734 min=26 avg=29
68: data copyin reached 1000 times
device time(us): total=23,888,588 max=33,546 min=23,378 avg=23,888
68: kernel launched 1000 times
grid: [1004] block: [256]
device time(us): total=2,398,101 max=2,961 min=2,137 avg=2,398
elapsed time(us): total=2,407,481 max=2,971 min=2,146 avg=2,407
68: data copyout reached 1000 times
device time(us): total=20,664,362 max=27,788 min=20,511 avg=20,664
77: data copyout reached 1000 times
device time(us): total=20,571,541 max=24,837 min=20,521 avg=20,571
```

Huge Data Transfer Bottleneck!
Computation: 5.19 seconds
Data movement: 74.7 seconds
Offloading a Parallel Kernel

CPU Memory

CPU

GPU Memory

GPU

PCle
Offloading a Parallel Kernel

For every parallel operation we:
1. Move the data from Host to Device
2. Execute once on the Device
3. Move the data back from Device to Host

What if we separate the data and execution?
Now we:
1. Move the data from Host to Device only when needed
2. Execute on the Device multiple times.
3. Move the data back from Device to Host when needed.
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]));
        }
    }
}

These copies happen every iteration of the outer while loop!

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.

```
!$acc data
  !$acc parallel loop
  ...
  !$acc parallel loop
  ...
!$acc end data
```

Arrays used within the data region 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
  ```c
  #pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
  ```

- Fortran
  ```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
Jacobi Iteration: OpenACC C Code

```c
#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++ ) {

            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++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator.
Did it help?

Set `PGI_ACC_TIME` environment variable to ‘1’

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

```
main NVIDIA devicenum=0
time(us): 4,802,950
51: data copyin reached 1 times
device time(us): total=22,768 max=22,768 min=22,768 avg=22,768
57: kernel launched 1000 times
grid: [4094] block: [256]
device time(us): total=2,611,387 max=2,817 min=2,593 avg=2,611
elapsed time(us): total=2,620,044 max=2,900 min=2,601 avg=2,620
57: reduction kernel launched 1000 times
grid: [1] block: [256]
device time(us): total=18,083 max=842 min=16 avg=18
elapsed time(us): total=27,731 max=852 min=25 avg=27
69: kernel launched 1000 times
grid: [4094] block: [256]
device time(us): total=2,130,162 max=2,599 min=2.112 avg=2.130
elapsed time(us): total=2,139,919 max=2,712 min=2
83: data copyout reached 1 times
device time(us): total=20,550 max=20,550 min=20,550 avg=20,550
```

0.23 seconds

0.24 seconds
Now OpenACC is 7.7X faster than 6 OpenMP threads and 18X faster than serial.
Further speedups

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

- By understanding more about the specific GPU on which you’re running, using these clauses may allow better performance.

- By understanding bottlenecks in the code via profiling, we can reorganize the code for even better performance.

- More on this in the Optimizing OpenACC session this afternoon.
Communication & IO with OpenACC
Calling MPI with OpenACC (Standard MPI)

```fortran
!$acc data copy(A)
!$acc parallel loop
do i=1,N
  ...
enddo
!$acc end parallel loop

call neighbor_exchange(A)

!$acc parallel loop
do i=1,N
  ...
enddo
!$acc end parallel loop
!$acc end data
```

Array “A” resides in GPU memory.

Routine contains MPI and requires “A.”

Array “A” returns to CPU here.
OpenACC update Directive

Programmer specifies an array (or partial array) that should be refreshed within a data region.

do_something_on_device()

!$acc update host(a)

do_something_on_host()

!$acc update device(a)

The programmer may choose to specify only part of the array to update.

Copy “a” from GPU to CPU

Copy “a” from CPU to GPU
Calling MPI with OpenACC (Standard MPI)

```fortran
$acc data copy(A)
$acc parallel loop
do i=1,N
  ...
enddo
$acc end parallel loop
$acc update host(A)
call neighbor_exchange(A)
$acc update device(A)
$acc parallel loop
do i=1,N
  ...
enddo
$acc end parallel loop
$acc end data
```

Copy “A” to CPU for MPI.

Return “A” after MPI to GPU.
OpenACC **host_data** Directive

Programmer specifies that host arrays should be used within this section, unless specified with **use_device**. This is useful when calling libraries that expect GPU pointers.

```c
!$acc host_data use_device(a)
call MPI_Sendrecv(a, ...)
!$acc end host_data

#pragma host_data use_device(a)
{
cublasDgemm(...,a,...);
}
```

This directive allows interoperability with a variety of other technologies, CUDA, accelerated libraries, OpenGL, etc.

Pass the device copy of “a” to subroutine.

Pass the device copy of “a” to function.
Calling MPI with OpenACC (GPU-aware MPI)

Pass device “A” directly to a GPU-aware MPI library called in \texttt{neighbor\_exchange}.

*More information about GPU-aware MPI libraries is available in other sessions, please see your agenda.*
OpenACC Tips & Tricks
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
- 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
Use time option to learn where time is being spent
  - (PGI) `PGI_ACC_TIME=1` (runtime environment variable)
  - (Cray) `CRAY_ACC_DEBUG=<1, 2, 3>` (runtime environment variable)
  - (CAPS) `HMPPRT_LOG_LEVEL=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
More OpenACC at GTC13

- S3019 - Tutorial: Optimizing OpenACC Codes - Monday 3/18 @ 14:30
- S3521 - Hands-on Lab: OpenACC Getting Started - Tuesday 3/19 @ 15:00
- S3532 - Hands-on Lab: OpenACC Data Management - Thursday 3/21 @ 14:00
- S3533 - Hands-on Lab: OpenACC Optimization - Thursday 3/21 @ 15:00

Plus several talks from our partners and customers, please see your agenda for more details.
Thank you