Introduction to OpenACC

2018 HPC Workshop: Parallel Programming

Alexander B. Pacheco
Research Computing
July 17 - 18, 2018
CPU vs GPU

**CPU**: consists of a few cores optimized for sequential serial processing

**GPU**: has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications
CPU vs GPU

CPU : consists of a few cores optimized for sequential serial processing

GPU : has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications
CPU vs GPU

CPU: consists of a few cores optimized for sequential serial processing
GPU: has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously
GPU enabled applications
CPU vs GPU

**CPU**: consists of a few cores optimized for sequential serial processing

**GPU**: has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously

GPU enabled applications
GPU Design

- Large number of cores working in SIMD mode.
- Slow global memory access, high bandwidth.
- CPU communication over PCI bus.
- Warp scheduling and fast switching queue model.
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Flexibility
GPU Accelerated Libraries

Some GPU-accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- ArrayFire
- CUSP
- Thrust
- IMLSL Library
- Sparse Linear Algebra
- C++ STL Features for CUDA
- Vector Signal Image Processing
- GPU Accelerated Linear Algebra
- Matrix Algebra on GPU and Multicore
GPU Programming Languages

### Numerical analytics
- MATLAB, Mathematica, LabVIEW

### Fortran
- OpenACC, CUDA Fortran

### C
- OpenACC, CUDA C

### C++
- CUDA C++, Thrust, Hemi, ArrayFire

### Python
- Anaconda Accelerate, PyCUDA, Copperhead

### .NET
- CUDAfy.NET, Alea.cuBase

developer.nvidia.com/language-solutions
What is OpenACC?

- OpenACC Application Program Interface describes a collection of compiler directive to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator.
- Provides portability across operating systems, host CPUs and accelerators

History

- OpenACC was developed by The Portland Group (PGI), Cray, CAPS and NVIDIA.
- PGI, Cray, and CAPs have spent over 2 years developing and shipping commercial compilers that use directives to enable GPU acceleration as core technology.
- The small differences between their approaches allowed the formation of a group to standardize a single directives approach for accelerators and CPUs.
- Full OpenACC 2.0 Specification available online
  - Implementations available now from PGI and Cray

The Standard for GPU Directives

Simple: Directive are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforwards and portable across parallel and multi-core processors

Powerful: GPU directives allow complete access to the massive parallel power of a GPU
What is OpenACC?

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
Why OpenACC

- Directives are easy and powerful.
- Avoid restructuring of existing code for production applications.
- Focus on expressing parallelism.

OpenACC is not GPU Programming

OpenACC is Expressing Parallelism in your code
SAXPY: Serial

**Serial Code**

```fortran
program saxpy

    implicit none
    integer, parameter :: dp = selected_real_kind(15)
    integer, parameter :: ip = selected_int_kind(15)
    integer(ip) :: i, n
    real(dp), dimension(:), allocatable :: x, y
    real(dp) :: a, start_time, end_time

    n = 5000000
    allocate(x(n), y(n))

    x = 1.0d0
    y = 2.0d0
    a = 2.0

    call cpu_time(start_time)
    do i = 1, n
        y(i) = y(i) + a * x(i)
    end do
    call cpu_time(end_time)
    deallocate(x, y)

    print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy
```

program saxpy

implicit none
integer, parameter :: dp = selected_real_kind(15)
integer, parameter :: ip = selected_int_kind(15)
integer(ip) :: i,n
real(dp),dimension(:), allocatable :: x, y
real(dp) :: a,start_time, end_time

n=500000000
allocate(x(n),y(n))
!$omp parallel sections
!$omp section
x = 1.0
!$omp section
y = 1.0
!$omp end parallel sections
a = 2.0

call cpu_time(start_time)
!$omp parallel do default(shared) private(i)
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$omp end parallel do
call cpu_time(end_time)
deallocate(x,y)

print '(a,f8.6)', 'SAXPY Time: ', end_time - start_time

drop program saxpy
OpenACC Code

```fortran
program saxpy
  
  use omp_lib
  
  implicit none
  integer :: i,n
  real,dimension(:), allocatable :: x, y
  real :: a, start_time, end_time

  n=500000000
  allocate(x(n), y(n))
  a = 2.0
  !$acc data create(x,y) copyin(a)
  !$acc parallel
  x(:) = 1.0
  !$acc end parallel
  !$acc parallel
  y(:) = 1.0
  !$acc end parallel

  start_time = omp_get_wtime()
  !$acc parallel loop
  do i = 1, n
    y(i) = y(i) + a * x(i)
  end do
  !$acc end parallel loop
  end_time = omp_get_wtime()
  !$acc end data
  deallocate(x, y)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'
end program saxpy
```
CUDA Fortran Code

```
module mymodule
contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  integer, parameter :: n = 100000000
  real, device :: x_d(n), y_d(n)
  real, device :: a_d
  real :: start_time, end_time

  x_d = 1.0
  y_d = 2.0
  a_d = 2.0

  call cpu_time(start_time)
  call saxpy<<<4096, 256>>>(n, a, x_d, y_d)
  call cpu_time(end_time)

  print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'
end program main
```
## Compile

## Speed Up

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Device</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Xeon E5-2650 v3</td>
<td>0.456777</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xeon E5-2670 v3</td>
<td>0.451826</td>
<td></td>
</tr>
<tr>
<td>OpenMP (20 threads)</td>
<td>Xeon E5-2650 v3</td>
<td>0.111399</td>
<td>4.100</td>
</tr>
<tr>
<td></td>
<td>Xeon E5-2670 v3</td>
<td>0.103316</td>
<td>4.373</td>
</tr>
<tr>
<td>OpenACC</td>
<td>Tesla K80</td>
<td>0.039771</td>
<td>11.775</td>
</tr>
<tr>
<td></td>
<td>GTX 1080</td>
<td>0.025179</td>
<td>18.260</td>
</tr>
<tr>
<td>CUDA</td>
<td>Tesla K80</td>
<td>0.026980</td>
<td>17.357</td>
</tr>
<tr>
<td></td>
<td>GTX 1080</td>
<td>0.019904</td>
<td>23.099</td>
</tr>
</tbody>
</table>
OpenACC Execution Model

- Application code runs on the CPU (sequential, shared or distributed memory).
- OpenACC directives indicate that the following block of compute intensive code needs to be offloaded to the GPU or accelerator.

```
!$acc end parallel
!$acc parallel
```

- GPU
- CPU

- Generate GPU code
- Allocate GPU memory
- Execute GPU code
- Copy input data
- Copy output data
- !$acc parallel
- !$acc end parallel
Building Block of OpenACC

- Program directives
  - Syntax
    - C/C++: #pragma acc <directive> [clause]
    - Fortran: !$acc <directive> [clause]
  - Regions
  - Loops
  - Synchronization
  - Data Structure
  - ...

- Runtime library routines
Clauses

- if (condition)
- async (expression)
- data management clauses
  - copy(···), copyin(···), copyout(···)
  - create(···), present(···)
  - present_or_copy{,in,out}(···) or pcopy{,in,out}(···)
  - present_or_create(···) or pcreate(···)
- reduction(operator:list)
Runtime Libraries

- System setup routines
  - acc_init(acc_device_nvidia)
  - acc_set_device_type(acc_device_nvidia)
  - acc_set_device_num(acc_device_nvidia)

- Synchronization routines
  - acc_async_wait(int)
  - acc_async_wait_all()
OpenACC kernels directive

C:  #pragma acc kernels [clause]

Fortran  !$acc kernels [clause]

- The kernels directive expresses that a region may contain parallelism and the compiler determines what can be safely parallelized.
- The compiler breaks code in the kernel region into a sequence of kernels for execution on the accelerator device.
- For the codes on the right, the compiler identifies 2 parallel loops and generates 2 kernels.
- What is a kernel? A function that runs in parallel on the GPU.
- When a program encounters a kernels construct, it will launch a sequence of kernels in order on the device.

```c
#include <stdio.h>

int main()
{
    int n = 10;
    double x[n], y[n];
    
    // Initial values
    for (int i = 0; i < n; i++)
    {
        x[i] = 1.0;
        y[i] = 2.0;
    }
    
    // Parallel loops
    #pragma acc kernels
    {
        for (int i = 0; i < n; i++)
        {
            y[i] = a * x[i] + y[i];
        }
    }
    
    return 0;
}
```

```fortran
program openacc_kernels

!$acc kernels
    do i = 1, n
        x(i) = 1.0
        y(i) = 2.0
    end do

    do i = 1, n
        y(i) = y(i) + a * x(i)
    end do

!$acc end kernels

!$acc kernels
    for (i = 0; i < n; i++)
    {
        x[i] = 1.0;
        y[i] = 2.0;
    }

    for (i = 0; i < n; i++)
    {
        y[i] = a * x[i] + y[i];
    }

end program openacc_kernels
```


The **parallel** directive identifies a block of code as having parallelism.

Compiler generates a parallel kernel for that loop.

**C:**
```
#pragma acc parallel [clauses]
```

**Fortran:**
```
!$acc parallel [clauses]
```

```c
C:
#include <stdio.h>

int main()
{
    int i, n;
    double x[n], y[n];

    // Initialization
    for (i = 0; i < n; i++)
    {
        x[i] = 1.0;
        y[i] = 2.0;
    }

    // Loop for computing y[i]
    for (i = 0; i < n; i++)
    {
        y[i] = a * x[i] + y[i];
    }

    return 0;
}
```

```fortran
Fortran:
!$acc parallel [clauses]
```

```fortran
Fortran:
do i = 1, n
    x(i) = 1.0
    y(i) = 2.0
end do

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

```c
C:
#include <stdio.h>

int main()
{
    int i, n;
    double x[n], y[n];

    // Initialization
    for (i = 0; i < n; i++)
    {
        x[i] = 1.0;
        y[i] = 2.0;
    }

    // Loop for computing y[i]
    for (i = 0; i < n; i++)
    {
        y[i] = a * x[i] + y[i];
    }

    return 0;
}
```
OpenACC Loop Directive

- Loops are the most likely targets for Parallelizing.
- The Loop directive is used within a parallel or kernels directive identifying a loop that can be executed on the accelerator device.

C:
```c
#pragma acc loop [clauses]
```

Fortran:
```fortran
!$acc loop [clauses]
```

- The loop directive can be combined with the enclosing parallel or kernels

C:
```c
#pragma acc kernels loop [clauses]
```

Fortran:
```fortran
!$acc parallel loop [clauses]
```

- The loop directive clauses can be used to optimize the code. This however requires knowledge of the accelerator device.

Clauses: gang, worker, vector, num_gangs, num_workers
OpenACC parallel vs. kernels

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

Both approaches are equally valid and can perform equally well.
program saxpy

use omp_lib

implicit none
integer :: i,n
real,dimension(:),allocatable :: x, y
real :: a,start_time, end_time

n=500000000
allocate(x(n),y(n))
a = 2.0
x(:) = 1.0
y(:) = 1.0

start_time = omp_get_wtime()
!$acc parallel loop
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end parallel loop
end_time = omp_get_wtime()
deallocate(x,y)

print '(a,f15.6)', 'SAXPY Time: ', end_time - start_time

end program saxpy

#include <stdio.h>
#include <time.h>
#include <omp.h>

int main() {
    long long int i, n=500000000;
    float a=2.0;
    float x[n];
    float y[n];
    double start_time, end_time;

    a = 2.0;
    for (i = 0; i < n; i++){
        x[i] = 1.0;
        y[i] = 2.0;
    }

    start_time = omp_get_wtime();
    #pragma acc kernels loop
    { 
        for (i = 0; i < n; i++){
            y[i] = a*x[i] + y[i];
        }
    }
    end_time = omp_get_wtime();

    printf("SAXPY Time: %f\n", end_time - start_time);
}

#include <stdio.h>
#include <time.h>
#include <omp.h>
Compilation

- **C:**
  ```
  pgcc -acc [-Minfo=accel] [-ta=tesla:cc60 -Mcuda=kepler+] -o saxpyc_acc saxpy_acc.c
  ```

- **Fortran 90:**
  ```
  pgf90 -acc [-Minfo=accel] [-ta=tesla:cc60 -Mcuda=kepler+] -o saxpyf_acc saxpy_acc.f90
  ```

```
[alp514.sol-b501](1006): pgcc -acc -ta=tesla:cc60 -Mcuda=kepler+ -Minfo=accel -o saxpyc_acc saxpy_acc.c
main:
  20, Generating implicit copyout(x[:500000000],y[:500000000])
  21, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
  21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  28, Generating implicit copyin(x[:500000000])
      Generating implicit copy(y[:500000000])
  29, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
  29, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
[alp514.sol-b501](1007): pgfortran -acc -ta=tesla:cc60 -Mcuda=kepler+ -Minfo=accel -o saxpyf_acc saxpy_acc.f90
saxpy:
  17, Generating implicit copyout(x::,y::)
      Accelerator kernel generated
      Generating Tesla code
  18, !$acc loop vector(128) ! threadidx%x
  18, Loop is parallelizable
  20, Accelerator kernel generated
      Generating Tesla code
  26, Generating implicit copyin(x::)
      Generating implicit copy(y::)
      Accelerator kernel generated
      Generating Tesla code
  27, !$acc loop gang, vector(128) ! blockIdx%x threadIdx%x
```
Run OpenACC Code

<table>
<thead>
<tr>
<th>Execution</th>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>SpeedUp</td>
</tr>
<tr>
<td>Serial</td>
<td>0.660000</td>
<td>0.664236</td>
</tr>
<tr>
<td>OpenMP (12 Threads)</td>
<td>0.215059</td>
<td>3.255</td>
</tr>
<tr>
<td>OpenMP (24 Threads)</td>
<td>0.130821</td>
<td>3.297</td>
</tr>
<tr>
<td>OpenACC (GTX 1080)</td>
<td>1.664477</td>
<td>0.401</td>
</tr>
</tbody>
</table>

- What’s going with OpenACC code?
- Why even bother with OpenACC if performance is so bad?
Analyzing OpenACC Run Time

- The PGI compiler provides automatic instrumentation when `PGI_ACC_TIME=1` at runtime

```plaintext
[alp514.sol-b501](1008): PGI_ACC_TIME=1 ./saxpyc_acc
SAXPY Time: 2.423356

Accelerator Kernel Timing data
/home/alp514/sum2017/saxpy/nodataregion/saxpy_acc.c
main NVIDIA devicenum=0
time(us): 4,987,729
14: compute region reached 1 time
15: kernel launched 1 time
grid: [65535] block: [128]
device time(us): total=30.948 max=30,948 min=30,948 avg=30,948
elapsed time(us): total=31.012 max=31,012 min=31,012 avg=31,012
14: data region reached 2 times
20: data copyout transfers: 478
device time(us): total=3,454,420 max=7,790 min=3,324 avg=7,226
22: compute region reached 1 time
23: kernel launched 1 time
grid: [65535] block: [128]
device time(us): total=50.330 max=50,330 min=50,330 avg=50,330
elapsed time(us): total=50.392 max=50,392 min=50,392 avg=50,392
22: data region reached 2 times
22: data copyin transfers: 478
device time(us): total=661.261 max=1,594 min=573 avg=1,383
26: data copyout transfers: 239
device time(us): total=790.770 max=3,809 min=1,327 avg=3,308

[alp514.sol-b501](1063): PGI_ACC_TIME=1 ./saxpyf_acc
SAXPY Time: 2.397494

Accelerator Kernel Timing data
/share/ceph/hpc2017/alp514/sum2017/openmp_acc/saxpy/nodataregion/saxpy_acc.f90
saxpy NVIDIA devicenum=0
time(us): 5,063,174
17: compute region reached 1 time
17: kernel launched 1 time
grid: [1] block: [128]
device time(us): total=154.492 max=154,492 min=154,492 avg=154,492
elapsed time(us): total=154.570 max=154,570 min=154,570 avg=154,570
17: data region reached 2 times
19: data copyout transfers: 478
device time(us): total=3,428.252 max=13,008 min=2,909 avg=7,172
20: compute region reached 1 time
20: kernel launched 1 time
grid: [1] block: [1]
device time(us): total=2 max=2 min=2 avg=2
elapsed time(us): total=53 max=53 min=53 avg=53
26: compute region reached 1 time
26: kernel launched 1 time
grid: [65535] block: [128]
device time(us): total=50.350 max=50,350 min=50,350 avg=50,350
elapsed time(us): total=50.402 max=50,402 min=50,402 avg=50,402
26: data region reached 2 times
26: data copyin transfers: 478
device time(us): total=658.588 max=1,536 min=577 avg=1,377
30: data copyout transfers: 239
device time(us): total=771.490 max=3,637 min=1,393 avg=3,227
```
Offloading a Parallel Kernel

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

What if we separate the data and Execution?
Offloading a Parallel Kernel

Now

1: Move data from Host to Device only when needed
2: Execute multiple times on the Device
3: Move data back from Device to Host when needed
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.

```c
!$acc data [clause]
!$acc parallel loop
...
!$acc end parallel loop
...
!$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.

- **Other clauses:** `present_or_copy[in|out]`, `present_or_create`, `deviceptr`. 
Array Shaping

- Compiler sometime cannot determine size of arrays
  - Must specify explicitly using the data clauses and array "shape"

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

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

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

- 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.
- Fortran
  ```fortran
  !$acc update [clause ...]
  ```
- C
  ```c
  #pragma acc update [clause ...]
  ```
- Clause
  - `host(list)`
  - `device(list)`
  - `if(expression)`
  - `async(expression)`
program saxpy

use omp_lib

implicit none

integer :: i, n
real, dimension(:), allocatable :: x, y
real :: a, start_time, end_time

n = 500000000
allocate(x(n), y(n))
a = 2.0
!$acc data create(x,y) copyin(a)
!$acc parallel
x(:) = 1.0
!$acc end parallel
!$acc parallel
y(:) = 1.0
!$acc end parallel

start_time = omp_get_wtime()
!$acc parallel loop
do i = 1, n
  y(i) = y(i) + a * x(i)
end do
!$acc end parallel loop
end_time = omp_get_wtime()
!$acc end data
deallocate(x,y)

print '(a,f15.6,a)', 'SAXPY Time: ', end_time - start_time, 'in secs'

end program saxpy
SAXPY using data clause

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time</th>
<th>SpeedUp</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>0.660000</td>
<td></td>
<td>0.664236</td>
<td></td>
</tr>
<tr>
<td>OpenMP (12 Threads)</td>
<td>0.215059</td>
<td>3.255</td>
<td>0.216842</td>
<td>5.351</td>
</tr>
<tr>
<td>OpenMP (24 Threads)</td>
<td>0.130821</td>
<td>3.297</td>
<td>0.230112</td>
<td>3.107</td>
</tr>
<tr>
<td>OpenACC (GTX 1080)</td>
<td>0.050374</td>
<td>13.102</td>
<td>0.050122</td>
<td>13.252</td>
</tr>
</tbody>
</table>
Reduction clause is allowed on *parallel* and *loop* constructs

**Fortran**

```fortran
!$acc parallel reduction(operation: var)  
  structured block with reduction on var  
!$acc end parallel
```

**C**

```c
#pragma acc kernels reduction(operation: var) {  
  structured block with reduction on var  
}
```
Further Speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker and vector clauses
- By understanding more about 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.
General Principles: Finding Parallelism in Code

- (Nested) for/do loops are best for parallelization
- Large loop counts are best
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
  - Use subscripted arrays, rather than pointer-indexed arrays
- Data regions should avoid wasted bandwidth
  - Can use directive to explicitly control sizes
- Various annoying things can interfere with accelerated regions.
  - Function calls within accelerated region must be inlineable.
  - No IO
OpenACC: Is it worth it?

- High-level. No involvement of OpenCL, CUDA, etc
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.
Lecture derived from slides and presentations by

- Michael Wolfe, PGI
- Jeff Larkin, NVIDIA
- John Urbanic, PSC

Search for OpenACC presentations at the GPU Technology Conference Website for further study
Exercise 1: SAXPY

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines
  \[ y \leftarrow \alpha x + y \]
- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 1 Pseudo Code for SAXPY

```plaintext
program SAXPY
    n ← some large number
    x(1 : n) ← some number say, 1
    y(1 : n) ← some other number say, 2
    a ← some other number ,say, 3
    do i ← 1 ··· n
        yi ← yi + a * xi
    end do
end program SAXPY
```
Exercise 2: Calculate pi by Numerical Integration I

- We know that

\[ \int_{0}^{1} \frac{4.0}{(1 + x^2)} \, dx = \pi \]

- So numerically, we can approximate pi as the sum of a number of rectangles

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

Meadows et al, A “hands-on” introduction to OpenMP, SC09
Exercise 2: Calculate pi by Numerical Integration II

**Algorithm 2** Pseudo Code for Calculating Pi

```plaintext
program CALCULATE_PI
    step ← 1/n
    sum ← 0
    do i ← 0 · · · n
        x ← (i + 0.5) * step; sum ← sum + 4/(1 + x^2)
    end do
    pi ← sum * step
end program
```
Exercise 2: Calculate pi by Numerical Integration III

```fortran
program pi_omp

  implicit none
  integer, parameter :: dp=selected_real_kind(14)
  integer, parameter :: ip=selected_int_kind(15)
  integer(ip) :: i
  integer(ip), parameter :: n=10000000000

  real(dp) :: x,pi,sum,step,start_time,end_time
  integer, dimension(8) :: value

  sum = 0d0
  step = 1.d0/float(n)
  !$acc data copyin(step) copyout(sum)
  call date_and_time (VALUES=value)
  start_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  !$acc parallel loop private(i,x) reduction(+:sum)
  do i = 0, n
    x = (i + 0.5d0) * step
    sum = sum + 4.d0 / (1.d0 + x ** 2)
  end do
  !$acc end parallel loop
  pi = step * sum
  call date_and_time (VALUES=value)
  end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  if (start_time > end_time) end_time = end_time + 3600d0
  !$acc end data

  print '(a,f17.15,f17.15)', 'pi = ', pi
  print '(a,f9.3,a)', 'time to compute = ', end_time - start_time, ' seconds'
end program pi_omp
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
  long long int i, n=10000000000;
  double start_time, end_time;
  double x, pi;
  double sum = 0.0;
  double step = 1.0/(double) n;

  #pragma acc data copyin(step) copyout(sum)
  {
    start_time = omp_get_wtime();
    #pragma acc parallel loops private(i,x) reduction(+:sum)
    {
      for (i = 0; i < n; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
      }
    }
    pi = step * sum;
    end_time = omp_get_wtime();
  }
  printf("pi = %17.15f\n",pi);
  printf("time to compute = %g seconds\n", (double)(end_time - start_time));
  return 0;
}
```

Exercise 3: Matrix Multiplication I

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix $C$ which is a product of two matrices $A$ and $B$:
  
  Element $i,j$ of $C$ is the dot product of the $i^{th}$ row of $A$ and $j^{th}$ column of $B$.

- Write a MATMUL code to multiple two matrices.
Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  
m, n ← some large number ≤ 1000
Define \( a_{mn}, b_{nm}, c_{mm} \)

\( a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0 \)

do \( i \leftarrow 1 \cdots m \)
  
do \( j \leftarrow 1 \cdots m \)
    
    \( c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} \ast b_{k,j} \)
    
  end do
end do

end program MATMUL
Exercise 3: Matrix Multiplication III

program matrix_mul
  implicit none
  integer, parameter :: dp = selected_real_kind(14)
  integer :: i,j,k
  integer, parameter :: nra=1500, nca=2000, ncb=1000
  real(dp) :: a(nra,nca) , b(nca,ncb) , c(nra,ncb)
  real(dp) :: flops,
  sum
  real(dp) :: init_time, start_time, end_time
  integer :: c1, c2, c3, cr
  integer, dimension(8) :: value
  flops = 2d0 * float(nra) * float(nca) * float(ncb)
!$acc data create(a,b,c)
call date_and_time (VALUES=value)
init_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  c = 0d0
  do 
    i = 1,nra
    do 
      j = 1,nca
      a(i,j) = i + j
    end do
  end do
  do 
    i = 1,nca
    do 
      j = 1,ncb
      b(i,j) = i * j
    end do
  end do
call date_and_time (VALUES=value)
start_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
!$acc parallel loop private(sum)
  do j = 1, nca
    do k = 1, ncb
      sum = 0d0
      !$acc loop reduction(+:sum)
      do i = 1, nra
        sum = sum + a(i,j) * b(j,k)
      end do
      c(i,k) = sum
    end do
  end do
!$acc end parallel loop
  call date_and_time (VALUES=value)
end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
!$acc end data
print ('(a,f6.3,a,f6.3,a,f7.3)'), 'Init Time: ', start_time - init_time, &
' Calc Time: ', end_time - start_time, &
' GFlops: ', 1d-9 + flops/(end_time - start_time)
end program matrix_mul