Introduction to OpenACC

Alexander B. Pacheco

User Services Consultant
LSU HPC & LONI
sys-help@loni.org

LONI Parallel Programming Workshop
Louisiana State University
Baton Rouge
June 10-12, 2013
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
**OpenACC I**

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 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 heterogenous 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
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))

  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

end program saxpy
OpenACC Code

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

```fortran
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

```
[apacheco@philip038 2013-LONI]$ pgf90 -o saxpy saxpy.f90
[apacheco@philip038 2013-LONI]$ pgf90 -mp -o saxpy_omp saxpy_omp.f90
[apacheco@philip038 2013-LONI]$ pgf90 -acc -ta=nvidia -o saxpy_acc saxpy_acc.f90
[apacheco@philip038 2013-LONI]$ pgf90 -o saxpy_cuda saxpy.cuf
```

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 X5650</td>
<td>0.231282</td>
<td>1</td>
</tr>
<tr>
<td>OpenMP (12 threads)</td>
<td>Xeon X5650</td>
<td>0.063231</td>
<td>3.6x</td>
</tr>
<tr>
<td>OpenACC</td>
<td>M2070</td>
<td>0.014329</td>
<td>16.1x</td>
</tr>
<tr>
<td>CUDA</td>
<td>M2070</td>
<td>0.006901</td>
<td>33.5x</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.

1. GPU
2. Generate GPU code
3. Allocate GPU memory
4. copy input data
5. Execute GPU code
6. copy output data
7. CPU
8. !$acc parallel
9. !$acc end parallel
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()`
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 contract, it will launch a sequence of kernels in order on the device.
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
!$acc parallel
do i = 1, n
   x(i) = 1.0
   y(i) = 2.0
end do
do i = 1, n
   y(i) = y(i) + a * x(i)
end do
!$acc end parallel

#pragma acc parallel
{
   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];
   }
}
```
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:  #pragma acc loop [clauses]
Fortran:  !$acc loop [clauses]

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

C:  #pragma acc kernels loop [clauses]
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

```c
#pragma acc loop
    do i = 1, n
        y(i) = y(i) + a * x(i)
    end do
#pragma acc end loop

#pragma acc kernels loop
for (i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
```

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

$acc kernels loop
for (i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
```
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 is safe.
- Can cover larger area of code with single directive.

Both approaches are equally valid and can perform equally well.
Compilation

- C:
  
  pgcc -acc [-Minfo=accel] [-ta=nvidia] -o saxpy_acc saxpy_acc.c

- Fortran 90:
  
  pgf90 -acc [-Minfo=accel] [-ta=nvidia] -o saxpyf_acc saxpy_acc.f90

Compiler Output

[apacheco@mikel nodataregion]$ pgcc -acc -ta=nvidia,time -Minfo=accel -o saxpy_acc saxpy_acc.c
main:
  19, Generating copyin(x[0:500000000])
  Generating copy(y[0:500000000])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  21, Loop is parallelizable
  Accelerator kernel generated
  21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  CC 1.0 : 10 registers; 44 shared, 0 constant, 0 local memory bytes
  CC 2.0 : 15 registers; 0 shared, 60 constant, 0 local memory bytes

[apacheco@mikel nodataregion]$ pgf90 -acc -ta=nvidia,time -Minfo=accel -o saxpyf_acc saxpy_acc.f90
saxpy:
  17, Accelerator kernel generated
  17, CC 1.0 : 7 registers; 40 shared, 4 constant, 0 local memory bytes
  CC 2.0 : 15 registers; 0 shared, 56 constant, 0 local memory bytes
  18, !$acc loop gang, vector(256) ! blockIdx%x threadIdx%x
  17, Generating copy(y(1:500000000))
  Generating copyin(x(1:500000000))
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary

[apacheco@mikel nodataregion]$
The PGI compiler provides automatic instrumentation when `PGI_ACC_TIME=1` at runtime.

You can also obtain automatic instrumentation by using the `-ta=nvidia,time` compiler flag.

```bash
$ ./saxpy_acc
```

```bash
Accelerator Kernel Timing data
/work/apacheco/2013-LONI/openmp/saxpy/nodataregion/saxpy_acc.c
main
  19: region entered 1 time
time(us): total=9,195,380 init=7,246,895 region=1,948,485
     kernels=58,028 data=1,835,336
w/o init: total=1,948,485 max=1,948,485 min=1,948,485 avg=1,948,485
  21: kernel launched 1 times
grid: [65535] block: [128]
time(us): total=58,028 max=58,028 min=58,028 avg=58,028
SAXPY Time: 9.195481
```

```bash
$ ./saxpyf_acc
```

```bash
Accelerator Kernel Timing data
/work/apacheco/2013-LONI/openmp/saxpy/nodataregion/saxpy_acc.f90
saxpy
  17: region entered 1 time
time(us): total=9,180,978 init=7,254,065 region=1,926,913
     kernels=59,013 data=1,923,915
w/o init: total=1,926,913 max=1,926,913 min=1,926,913 avg=1,926,913
  17: kernel launched 1 times
grid: [65535] block: [256]
time(us): total=59,013 max=59,013 min=59,013 avg=59,013
SAXPY Time: 9.181015
```
What’s going with OpenACC code?

Why even bother with OpenACC if performance is so bad?
Offloading a Parallel Kernel

CPU Memory

GPU Memory

CPU

GPU

PCIe
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
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.
Compiler sometime cannot determine size of arrays

- Must specify explicitly using the 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:size)), copyout(b(s/4:3*s/4))
```

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

\$acc update [clause ...]

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
    !$acc end parallel
    !$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
## Exercise: Redo Matrix Multiplication Problem I

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time</th>
<th>SpeedUp</th>
<th>GFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial</strong></td>
<td>6.226</td>
<td></td>
<td>0.964</td>
</tr>
<tr>
<td>OpenMP 16 CPUs</td>
<td>0.444</td>
<td>14.022</td>
<td>13.03</td>
</tr>
<tr>
<td>OpenACC</td>
<td>0.175</td>
<td>35.577</td>
<td>34.265</td>
</tr>
</tbody>
</table>

### Fortran

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time</th>
<th>SpeedUp</th>
<th>GFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial</strong></td>
<td>7.113</td>
<td></td>
<td>0.844</td>
</tr>
<tr>
<td>OpenMP 16 CPUs</td>
<td>0.494</td>
<td>14.399</td>
<td>12.146</td>
</tr>
<tr>
<td>OpenACC</td>
<td>0.257</td>
<td>27.677</td>
<td>23.346</td>
</tr>
</tbody>
</table>
Exercise: Redo Matrix Multiplication Problem II

```fortran
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
      !$acc loop reduction(+:sum)
      do i = 1, nra
        sum = sum + a(i,j) * b(j,k)
      end do
      c(i,k) = sum
    end do
  end do
  call date_and_time(VALUES=value)
  end_time = float(value(6)*60) + float(value(7)) + float(value(8))/1000d0
  !$acc end parallel loop
  call date_and_time(VALUES=value)
  end_time - init_time, start_time, end_time
  integer, dimension(8) :: value

  print '(a,f6.3,a,f6.3,a,f7.3)'
end program matrix_mul
```

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

#define dt(start, end) ((end.tv_sec - start.tv_sec) + \n 1/1000000.0*(end.tv_usec - start.tv_usec))

int main() {
  int i,j,k;
  int nra=1500, nca=2000, ncb=1000;
  double a[nra][nca], b[nca][ncb], c[nra][ncb];
  struct timeval icalc, scale, ecalc;
  double flops, sum, timing;
  flops = 2.0 * nra * nca * ncb;

  #pragma acc data create(a,b,c)
  {
    gettimeofday(&icalc, NULL);
    for (i = 0; i < nra; i++)
      for (j = 0; j < nca; j++)
        a[i][j] = (double)(i+j);
  }

  for (j = 0; j < nca; j++)
    for (k = 0; k < ncb; k++)
      b[j][k] = (double)(i+j);

  for (i = 0; i < nra; i++)
    for (k = 0; k < ncb; k++)
      c[i][k] = 0.0;

  gettimeofday(&scale, NULL);
  #pragma acc parallel loop private(sum)
  for (i = 0; i < nra; i++)
    for (k = 0; k < ncb; k++)
      sum = 0.0;

  #pragma acc loop seq
  for (j = 0; j < nca; j++)
    for (k = 0; k < ncb; k++)
      c[i][k] = sum;

  gettimeofday(&ecalc, NULL);

  timing = dt(scale, ecalc);
  printf("Init Time: %6.3f Calc Time: %6.3f GFlops: %7.3f\n", dt(icalc, scale), timing, 1e-9*flops/timing );
}
```

Introduction to OpenACC June 10-12, 2013

HPC@LSU - http://www.hpc.lsu.edu 29 / 34
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
}
```

- Redo Calculation of Pi in OpenACC and compare timing results with serial and OpenMP.
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.
(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
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