EuroHack15: Introduction to OpenACC

Alistair Hart
ahart@cray.com

Contents

- **What is OpenACC?**
- **How do GPUs work?**
  - Will my code run well on a GPU using OpenACC?
- **What does OpenACC look like?**
- **How do I use it?**
  - The basic concepts
  - The basic directives
    - Advanced topics will be covered later in this tutorial
    - And in other training courses, like this one.
- **Plus a few hints, tips, tricks and gotchas along the way**
  - Not all guaranteed to be relevant, useful (or even true).

Accelerator programming

- **Why do we need a new GPU programming model?**
- **Aren't there enough ways to drive a GPU already?**
  - CUDA (incl. NVIDIA CUDA-C & PGI CUDA-Fortran)
  - OpenCL
- **All are quite low-level and closely coupled to the GPU**
  - User needs to rewrite kernels in specialist language:
    - Hard to write and debug
    - Hard to optimise for specific GPU
    - Hard to port to new accelerator
  - Multiple versions of kernels in codebase
    - Hard to add new functionality
- **Aside: OpenMP4accel is another directive-based model**
  - Currently lacks some features of OpenACC
  - Compiler support also currently immature

Directive-based programming

Directives provide a high-level alternative

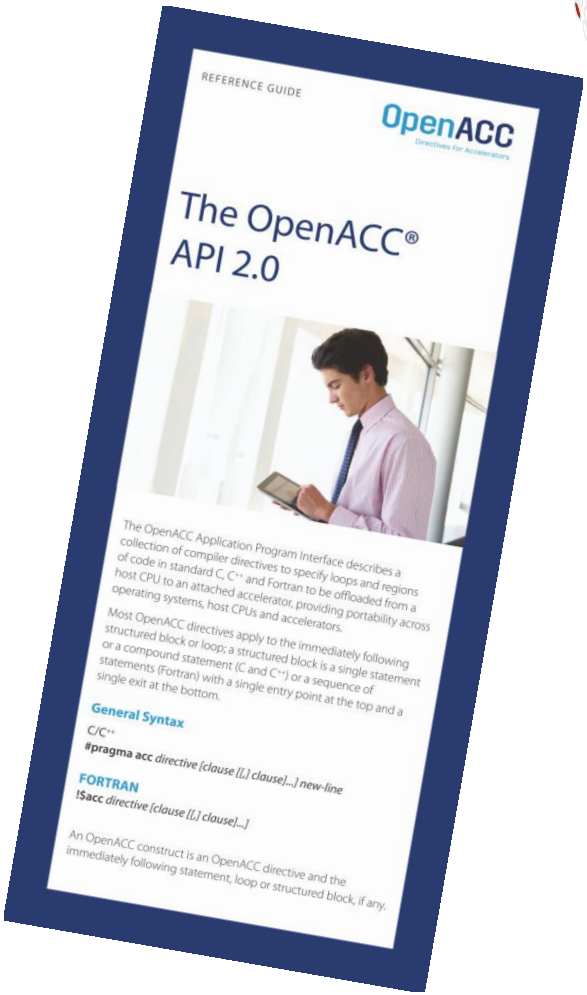
+ **Based on original source code (Fortran, C, C++)**
  + Easier to maintain/port/extend code
  + Users with OpenMP experience find it a familiar programming model
  + Compiler handles repetitive coding (cudaMalloc, cudaMemcpy...)
  + Compiler handles default scheduling; user tunes only where needed

– **Possible performance sacrifice**
  – Important to quantify this
  – Can then tune the compiler
  – Small performance sacrifice is acceptable
    – trading-off portability and productivity
    – after all, who handcodes in assembler for CPUs these days?

OpenACC
Directives for Accelerators

CRAY

- **A common directive programming model for today's GPUs**
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer:
      - portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at openacc.org
    - Initially implementations targeted at NVIDIA GPUs



- **Compiler support: all now complete**
  - Cray CCE: complete OpenACC 2.0 in v8.2
  - PGI Accelerator: v12.6 onwards
  - gcc: work started in late 2013
  - Various other compilers in development

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

Accelerator directives

- **Modify original source code with directives**
  - Non-executable statements (comments, pragmas)
    - Can be ignored by non-accelerating compiler
    - CCE -hnoacc also suppresses compilation
  - Sentinel: acc
    - C/C++: preceded by #pragma
      - Structured block {...} avoids need for end directives
    - Fortran: preceded by !$ (or c$ for FORTRAN77)
      - Usually paired with !$acc end * directive
      - Directives can be capitalized
  - Continuation to extra lines allowed
    - C/C++: \ (at end of line to be continued)
    - Fortran:
      - Fixed form: c$acc& or !$acc& on continuation line
      - Free form: & at end of line to be continued
        - continuation lines can start with either !$acc or !$acc&

```
// C/C++ example
#pragma acc *
{structured block}
```

```
! Fortran example
!$acc *
<structured block>
!$acc end *
```

Conditional compilation

- **In theory, OpenACC code should be identical to CPU**
  - only difference are the directives (i.e. comments)
- **In practise, you may need slightly different code**
  - For example, to cope with:
    - calls to OpenACC runtime API functions
    - where you need to recode for OpenACC
      - such as for performance reasons
      - you should try to minimise this
        - usually better OpenACC code is better CPU code
- **CPP macro defined to allow conditional compilation**
  - _OPENACC == yyyymm
    - Version 1.0: 201111
    - Version 2.0: 201306

A first example

- **Execute loop nest on GPU**

- **Compiler does the work:**
  - Data movement
    - determines data use in loopnest
    - at start and end of loopnest:
      - allocates/frees GPU memory
      - moves data to/from GPU
  - Synchronisation

```
#pragma acc parallel loop
for (j=0; j<N; j++) {
  for (i=1; i<N-1; i++) {
    c[j][i] = a[j][i] + b[j][i]
  }
}
```

write-only

read-only

  - Loop "schedule": spreading loop iterations over threads on GPU
    - OpenACC will "partition" (workshare) more than one loop in a loopnest
      - compare this to OpenMP, which only partitions the outer loop
  - Caching (e.g. explicit use GPU shared memory for reused data)
    - automatic caching can be important
  - User can tune all default behavior with optional clauses on directives

Accelerator kernels

- **We call a loopnest that will execute on the GPU a "kernel"**
  - this language is similar to CUDA
  - the loop iterations will be divided up and executed in parallel
- **We have choice of two directives to create a kernel**
  - parallel loop or kernels loop
    - both generate an accelerator computational task from a loopnest
      - also known as a "kernel"
      - the language is confusing
- **Why are there two and what's the difference?**
  - You can use either
    - or both, in different parts of the code
  - This tutorial concentrates on using the parallel loop directive

A first full OpenACC program: "Hello World"

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
  <stuff>
END PROGRAM main
```

- Two accelerator parallel regions
  - Compiler creates two kernels
    - Loop iterations automatically divided across GPU threads
  - First kernel initialises array
    - Compiler will determine a is write-only
  - Second kernel updates array
    - Compiler will determine a is read-write
  - Breaking parallel region=barrier
    - No barrier directive (global or within SM)

- Note:
  - Code can still be compiled for the CPU

Data scoping

- **Codes process data, using other data to do this**
  - all this data is held in structures, such as arrays or scalars

- **In a serial code (or pure MPI), there are no complications**
- **In a thread-parallel code (OpenACC, OpenMP etc.)**
  - Things are more complicated:
  - Some data will be the same for each thread (e.g. the main data array)
    - The threads can (and usually should) share a single copy of this data
  - Some data will be different (e.g. loop index values)
    - Each thread will need it's own private copy of this data

- **Data scoping arranges this. It is done:**
  - automatically (by the compiler) or explicitly (by the programmer)

- **If the data scoping is incorrect, we get:**
  - incorrect (and inconsistent) answers ("race conditions"), and/or
  - a memory footprint that is too large to run

Understanding data scoping

- **Data scoping ensures the right answer**
  - We want the same answer when executing in parallel as when serially

- **Declare variables in parallel region to be shared or private**
  - shared
    - all loop iterations process the same version of the variable
    - variable could be a scalar or an array
    - **a** and **b** are shared arrays in this example
  - private
    - each loop iteration uses the variable separately
    - again, variable could be a scalar or an array
    - **t** is a private scalar in this example
    - loop index variables (like **i**) are also private
  - firstprivate: a variation on private
    - each thread's copy set to initial value
    - loop limits (like **N**) should be firstprivate

```
for (i=0; i<N; i++) {
   t = a[i];
   t++;
   b[i] = 2*t;
}
```

Data scoping in OpenACC (and OpenMP)

- **In OpenMP, we have exactly these data clauses**
  - shared, private, firstprivate

- **In OpenACC**
  - private, firstprivate are just the same
  - shared variables are more complicated in OpenACC
    - because we also need to think about data movements to/from GPU
  - We sub-classify shared variables by how they are used on the GPU:
  - copyin: a shared variable that is used read-only by the GPU
  - copyout: a shared variable that is used write-only
  - copy: a shared variable that is used read-write
  - create: a shared variable that is a temporary scratch space (although there is still an unused copy on the host in this case)

Data scoping with OpenACC

- **parallel regions:**
  - scalars and loop index variables are private by default
  - arrays are shared by default
    - the compiler chooses which shared-type: copyin, copyout, etc.
  - explicit data clauses over-ride automatic scoping decisions

  - You can also add the default(none) clause
    - then you have to do everything explicitly (or you get a compiler error)

A more-explicit first version

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc parallel loop copyout(a)
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
!$acc parallel loop copy(a)
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
  <stuff>
END PROGRAM main
```

- **We could choose to make the data movements explicit**
  - maybe because we want to
    - maybe also use default(none) clause
  - or maybe compiler is overcautious

- Note:
  - Array a is needlessly moved from/to GPU between kernels
    - You could call this "data sloshing"
    - This will have a big impact on performance

OpenACC data regions

- **Data regions allow data to remain on the accelerator**
  - e.g. for processing by multiple accelerator kernels
  - specified arrays only move at start/end of data region
- **Data regions only label a region of code**
  - they do not define or start any sort of parallel execution
  - just specify GPU memory allocation and data transfers
  - can contain host code, nested data regions and/or device kernels
- **Be careful:**
  - Inside data region we have two copies of each of the specified arrays
  - These only synchronise at the start/end of the data region
    - and only following the directions of the explicit data clauses
  - Otherwise, you have two separate arrays in two separate memory spaces
  - Unified memory systems will have only one copy!

Defining OpenACC data regions

- **Two ways to define data regions:**
  - Structured data regions:
    - Fortran: !$acc data [data-clauses] ... !$acc end data
    - C/C++: #pragma acc data [date-clauses] {...}
  - Unstructured data regions (new in OpenACC v2):
    - Fortran: !$acc enter data [data-clauses] ... !$acc exit data [data-clauses]
    - C/C++: #pragma enter data [data-clauses] ... #pragma exit data [data-clauses]

- **For most "procedural code", use structured data regions**

- **Unstructured data regions**
  - Useful for more "Object Oriented" coding styles, e.g.
    - Separate constructor/destructor methods in C++
    - Separate subroutines for malloc (or allocate) and free (or deallocate)

- **A data region with no data clauses is pointless**
  - that is, it is redundant (and does nothing)

A second version

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copyout(a)
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
!$acc end data
  <stuff>
END PROGRAM main
```

- Now added a data region
  - Specified arrays only moved at boundaries of data region
  - Unspecified arrays moved by each kernel
  - No compiler-determined movements for data regions
- Data region can contain host code and accelerator regions
- Copies of arrays independent

- **No automatic synchronisation within data region**
  - User-directed synchronisation possible with update directive

Data scoping with OpenACC (2)

- **parallel regions:**
  - scalars and loop index variables are firstprivate by default
  - arrays are shared by default
    - the compiler chooses which shared-type: copyin, copyout, etc.
  - explicit data clauses over-ride automatic scoping decisions
    - You can also add the default(none) clause
    - then you have to do everything explicitly (or you get a compiler error)

- **data regions:**
  - only shared-type scoping clauses are allowed
  - there is NO default/automatic scoping
  - un-scoped variables on data regions
    - will be scoped at each of the enclosed parallel regions
      - automatically, unless the programmer does this explicitly
    - this probably leads to unwanted data-sloshing of large arrays
  - Using data region scoping in enclosed parallel regions:
    - same routine: omit scoping clauses on enclosed parallel directives
    - different routine: use present clause on enclosed parallel directives

Sharing GPU data between subprograms

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copyout(a)
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
  CALL double_array(a)
!$acc end data
  <stuff>
END PROGRAM main
```

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$acc parallel loop present(b)
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$acc end parallel loop
END SUBROUTINE double_array
```

```
INTEGER FUNCTION double_scalar(c)
  INTEGER :: c
  double_scalar = 2*c
END FUNCTION double_scalar
```

- **present clause uses GPU version of b without data copy**
  - Original calltree structure of program can be preserved
- **One kernel is now in subroutine (maybe in separate file)**
  - OpenACC 1.0: function calls inside parallel regions required inlining
  - OpenACC 2.0: compilers support calls and nested parallelism

Reduction variables

- **Reduction variables are a special case of shared variables**
  - where we will need to combine values across loop iterations
  - e.g. sum, max, min, logical-and etc. acting on a shared array

- **We need to tell the compiler to treat this appropriately**
  - Use the reduction clause for this (added to parallel loop directive)
    - same expression in OpenACC as in OpenMP
  - Examples:
    - sum: use clause reduction(+:t)
      - Note sum could involve adding and/or subtracting
    - max: use clause reduction(max:u)

```
DO i = 1,N
   t = t + a(i) - b(i)
   u = MAX(u,a(i))
ENDDO
```

- **Note: OpenACC only allows reductions of scalars**
  - not of array elements
  - advice:
    - try rewriting to use a temporary scalar in the loopnest for the reduction

Data scoping gotchas: OpenACC vs. OpenMP

- **In OpenACC parallel regions:**
  - scalars *and* loop index variables are firstprivate by default
- **Compare this to OpenMP parallel regions:**
  - loop index variables are private by default, but scalars are shared
- **Be careful of this, especially:**
  - if you program (separately) using the two programming models, or
  - if you are translating an OpenMP code to OpenACC

Minimising data movements

- **This is the single-biggest OpenACC optimisation for GPUs**

- **There are three techniques:**
  - Keep data on the GPU as long as possible
    - use data regions and port all enclosed loopnests (as we have seen)
  - Only move arrays when you need to
    - using the update directive
  - Only move the data you need to move
    - using array sections

The update directive

- **Data regions keep data on device**
  - can span multiple compute kernels and serial (host) code
  - create *copies* of data arrays on device for duration of data region
  - host, device copies only synchronised at start/end of data region
    - as requested by explicit data clauses
- **You can synchronise copies manually within a data region**
  - for instance:
    - to copy a halo buffer back to the host for communication
    - to copy values of an array to the CPU for checking or printing
- **You do this using the update directive, for instance:**
  - update host(a) copies entire array a from device to host
    - OpenACC 2.0: can use self instead of host
  - update device(a) copies entire array a from host to device

Array sections

- **Sometimes we only need to move part of an array**
  - array section notation allows this, using ":" notation
  - syntax differs slightly between languages
    - Fortran uses start:end, so first N elements is a(1:N)
    - C/C++ uses start:length, so first N elements is b[0:N]
    - Advice: be careful when switching languages!
      - Use profiler, CRAY_ACC_DEBUG commentary to see how much data moved

- **Sections allowed in data clauses and with update**

- **For multi-dimensional arrays**
  - specified sections must be a contiguous block of memory
  - can only specify one incomplete section on slowest-moving index
    - Fortran: slowest index is right-most
      - so a(1:N,2:N-1) is allowed, but a(2:N-1,1:N) is not
    - C/C++: slowest index is left-most
      - so b[1:N-2][0:N] is allowed, but b[0:N][1:N-2] is not

Unshaped pointers

- **"Unshaped pointer" compiler/runtime errors**
  - Fortran arrays have a complicated data structure
    - includes a descriptor that contains information about size and shape
    - the compiler therefore knows how much data to transfer
  - C/C++: arrays are often just pointers
    - especially if the arrays were dynamically allocated or passed by reference
    - How many bytes should be transferred here: copy(c) ?
    - So you usually need to be more explicit
      - You need to put in data clauses
      - And specify the slicing (even if this is the whole array): copy(c[0:N])

Sharing data between kernels

- **If you are using a data region around kernels**
  - Must ensure that the runtime uses the shared data already present
- **If the kernel is in the same routine as the data region**
  - just don't mention those bits of data in the parallel/kernels clauses
- **If the kernel is in a different routine to the data region**
  - On the parallel or kernels directive:
    - Specify the relevant data with present clause
    - instead of other shared clauses (copy, copyin, copyout, create)
    - don't rely on automatic scoping for shared data in this case
- **If an array is declared present, but is not on accelerator**
  - you get a runtime error and the program crashes
  - This is usually what you would like to happen
    - rather than running to completion with the wrong answer
  - But there are other ways to do things...

Directives in summary

- **Compute regions**
  - created using parallel loop or kernels loop directives
- **Data regions**
  - created using data or enter/exit data directives
- **Data clauses are applied to:**
  - accelerated loopnests: parallel and kernels directives
    - here they over-ride relevant parts of the automatic compiler analysis
    - you can switch off all automatic scoping with default(none) clause (in v2)
  - data regions: data directive (plus enter/exit data in OpenACC v2)
    - There is no automatic scoping in data regions (arrays or scalars)
  - Shared clauses (copy, copyin, copyout, create)
    - supply list of scalars, arrays (or array sections)
  - Private clauses (private, firstprivate, reduction)
    - only apply to accelerated loopnests (parallel and kernels directives)
  - present clause (used for nested data/compute regions)

And take a breath...

- **You now know everything you need to start accelerating**
  - You can successfully port a lot of codes just knowing this much
  - The performance at this stage isn't bad, either
    - you can often beat the CPU version of the code running across all the cores
- **So what is the rest of OpenACC (and this tutorial) for?**
  - Some codes require more functionality to port
  - OpenACC also has a lot of performance tuning options
- **The emphasis in this introduction has been on**
  - explaining data scoping and using data regions
- **Why?**
  - because optimising data movements is far more important than tuning
    - minimising data transfers typically speeds up GPU execution by 10x-100x
    - performance tuning maybe gains you 2x-3x
      - and you can't start to get this until you first stop data-sloshing