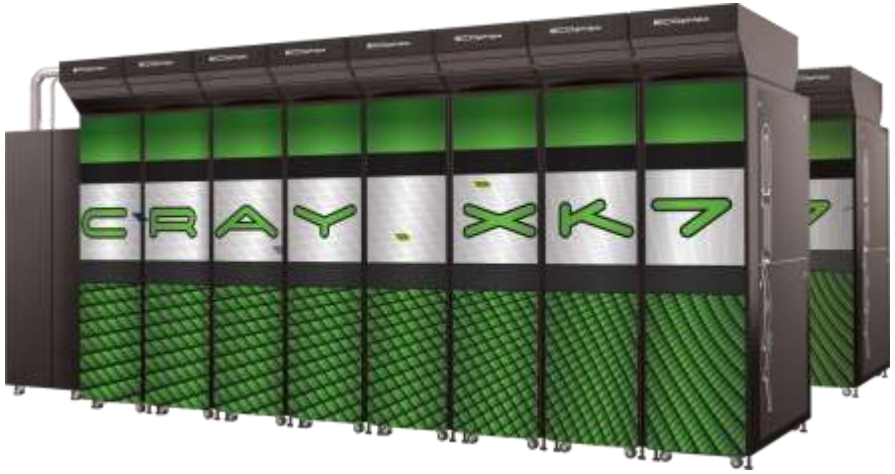
The OpenACC programming model

Alistair Hart
Cray Exascale Research Initiative Europe



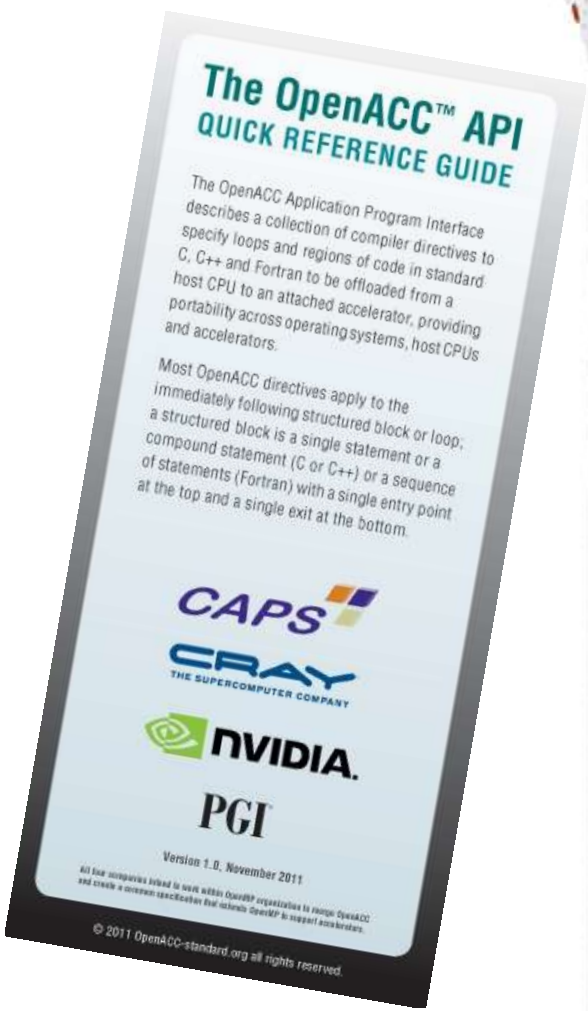
Contents

- **What is OpenACC?**
- **How does it work?**
  - The execution and memory models
- **What does it looks like?**
- **How do I use it?**
  - Basic directives
    - Enough to do the first two practicals
    - Advanced topics will follow in another lecture
- **Where can I learn more?**

- **Plus a few hints, tips, tricks and gotchas along the way**
  - Not all guaranteed to be relevant, useful (or even true)

OpenACC
DIRECTIVES FOR ACCELERATORS

- **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
- **Current version: 1.0 (November 2011)**
  - v2.0 expected in 1H 2013
- **Compiler support: all now complete**
  - Cray CCE: complete in 8.1 release
  - PGI Accelerator: version 12.6 onwards
  - CAPS: Full support in v1.3
  - (accULL: research compiler, C only)



CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

OpenMP accelerator directives

- **A common programming model for tomorrow's accelerators**
- **An established open standard is the most attractive**
  - portability; multiple compilers for debugging; permanence
- **Currently with subcommittee of OpenMP ARB**
  - includes most major vendors + others (e.g. EPCC)
  - co-chaired by Cray (James Beyer) and TI (Eric Stotzer)
  - aiming for OpenMP 4.0
- **Targets Fortran, C, C++**
- **Current version: draft**

- **OpenACC will continue to be supported**
  - Developers can transition to OpenMP if they wish
  - Converting OpenACC to OpenMP will be straightforward

OpenACC Execution model

- **In short:**
  - It's just like CUDA

OpenACC Execution model

- **In detail:**
  - Host-directed execution with attached GPU accelerator
  - Main program executes on “host” (i.e. CPU)
    - Compute intensive regions offloaded to the accelerator device
    - under control of the host.
  - “device” (i.e. GPU) executes parallel regions
    - typically contain “kernels” (i.e. work-sharing loops), or
    - kernels regions, containing one or more loops which are executed as kernels.
  - Host must orchestrate the execution by:
    - allocating memory on the accelerator device,
    - initiating data transfer,
    - sending the code to the accelerator,
    - passing arguments to the parallel region,
    - queuing the device code,
    - waiting for completion,
    - transferring results back to the host, and
    - deallocating memory.
  - Host can usually queue a sequence of operations
    - to be executed on the device, one after the other.

OpenACC Memory model

- **In short:**
  - it's just like CUDA

OpenACC Memory model

- **In detail:**
  - Memory spaces on the host and device distinct
    - Different locations, different address space
    - Data movement performed by host using runtime library calls that explicitly move data between the separate
  - GPUs have a weak memory model
    - No synchronisation between different execution units (SMs)
      - Unless explicit memory barrier
    - Can write OpenACC kernels with race conditions
      - Giving inconsistent execution results
      - Compiler will catch most errors, but not all (no user-managed barriers)
  - OpenACC
    - data movement between the memories implicit
      - managed by the compiler,
      - based on directives from the programmer.
    - Device memory caches are managed by the compiler
      - with hints from the programmer in the form of directives.

Accelerator directives

- **Modify original source code with directives**
  - Non-executable statements (comments, pragmas)
    - Can be ignored by non-accelerating compiler
    - CCE -hnoacc (or -xacc) also supresses 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 !$acc end *
      - Directives can be capitalised
  - Continuation to extra lines allowed
    - C/C++: \ (at end of line to be continued)
    - Fortran:
      - Fixed form: c$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**
  - E.g.
    - around calls to OpenACC runtime API functions
    - where you need to recode for OpenACC, e.g. for performance reasons
      - try to minimise this; usually better OpenACC code is better CPU code
- **CPP macro defined to allow conditional compilation**
  - _OPENACC == yyyymm (currently 201111)

A first example

Execute a loop nest on the GPU

- **Compiler does the work:**
  - Data movement
    - allocates/frees GPU memory at start/end of region
    - moves of data to/from GPU

```
!$acc parallel loop
DO j = 1,N
 DO i = 2,N-1
    c(i,j) = a(i,j) + b(i,j)
  ENDDO
ENDDO
!$acc end parallel loop
```

write-only

read-only

  - Loop schedule: spreading loop iterations over PEs of GPU
    - **OpenACC** **CUDA**
    - `gang`: a threadblock
    - `worker`: warp (group of 32 threads)
    - `vector`: threads within a warp
    - Compiler takes care of cases where iterations doesn't divide threadblock size
  - Caching (explicitly use GPU shared memory for reused data)
    - automatic caching (e.g. NVIDIA Fermi, Kepler) important
  - Tune default behaviour with optional clauses on directives

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 gangs, workers, vectors
    - Breaking parallel region acts as barrier
  - First kernel initialises array
    - Compiler will determine copyout(a)
  - Second kernel updates array
    - Compiler will determine copy(a)
  - Breaking parallel region=barrier
    - No barrier directive (global or within SM)

- **Array a(:) unnecessarily moved from and to GPU between kernels**
  - "data sloshing"
- **Code still compile-able for CPU**

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 of copies within data region**
  - User-directed synchronisation via update directive

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
```

- **One of the kernels now in subroutine (maybe in separate file)**
  - CCE supports function calls inside parallel regions
    - Fermi: Compiler will inline (maybe need `-Oipafrom` or program library)
- **present clause uses version of b on GPU without data copy**
  - Can also call double_array() from outside a data region
    - Replace present with present_or_copy
- **Original calltree structure of program can be preserved**

Data clauses

- **Applied to: data, parallel [loop], kernels [loop]**
  - copy, copyin, copyout
    - copy moves data "in" to GPU at start of region and/or "out" to CPU at end
    - supply list of arrays or array sections (using ":" notation)
    - N.B. Fortran uses start:end; C/C++ uses start:length
      - e.g. first N elements: Fortran 1:N (familiar); C/C++ 0:N (less familiar)
      - Advice: be careful and don't make mistakes!
      - Use profiler and/or runtime commentary to see how much data moved
      - Avoid non-contiguous array slices for performance
  - create
    - No copyin/out – useful for shared temporary arrays in loopnests
    - Host copy still exists
  - private, firstprivate: as per OpenMP
    - scalars private by default (not just loop variables)
    - Advice: declare them anyway, for clarity

More data clauses

- present, present_or_copy*, present_or_create
  - pcopy*, pcreate for short
  - Checks if data is already on the device
    - if it is, it uses that version
      - no data copying will be carried out for that data
    - if not, it does the prescribed data copying
  - Advice: only use present_or_* if you really have to
    - "not present" runtime errors are a useful development tool for most codes

- In both cases, the data is processed on the GPU
- Advanced topic: what if I want to call routine either:
  - with data on the GPU, to be processed on the GPU, or...
  - with data on the CPU, to be processed on the CPU?
- Either:
  - Explicitly call one of two versions of the routine, one with OpenACC, or...
  - Use the Cray OpenACC runtime to check if data present and branch code

And take a breath...

- **You now know everything you need to start accelerating**
  - This is all you need to do Practical 1 and Practical 2
  - Just using what you know, the code in Practical 2:
    - is fully ported to the GPU
    - runs faster on the GPU than it does across 16 cores of the CPU
- **So what do we do for the rest of the lecture (let alone the rest of the day)?**
  - Not all codes are as simple as Practical 2
  - OpenACC has a lot more functionality to cover
  - And we want to be able to tune the performance

Clauses for !$acc parallel loop

- **Tuning clauses:**
  - !$acc loop [gang] [worker] [vector]
    - Targets specific loop (or loops with collapse) at specific level of hardware
      - gang ↔ CUDA threadblock (scheduled on a single SM)
      - worker ↔ CUDA warp of 32 threads (scheduled on vector unit)
      - vector↔ CUDA threads in warp executing in SIMT lockstep
    - You can specify more than one
      - !$acc loop gang worker vector schedules loop iteration over all hardware
    - We'll discuss loop scheduling in much more detail later

More clauses for !$acc parallel loop

- **More tuning clauses:**
- **num_gangs, num_workers, vector_length**
  - Tunes the amount of parallelism used (threadblocks, threads/block...)
  - To set the number of threads per block (fixed at compile time for CCE)
    - vector_length(NTHREADS) *or* num_workers(NTHREADS/32)
    - NTHREADS must be one of: 1, 64, 128 (default), 256, 512, 1024
    - NTHREADS > 32 automatically decomposed into warps of length 32
  - Don't need to specify number of threadblocks (unless you want to)
  - Handy tip: To debug a kernel by running on a single GPU thread, use:
    - !$acc parallel [loop] gang vector num_gangs(1) vector_length(1)
    - Useful for checking race conditions in parallelised loopnests (but very slow)

More OpenACC directives

- **Other !$acc parallel loop clauses:**
  - seq: loop executed sequentially
  - independent: compiler hint, if it isn't partitioning (parallelising) a loop
  - if(logical)
    - Executes on GPU if .TRUE. at runtime, otherwise on CPU
  - reduction: as in OpenMP
  - cache: specified data held in software-managed data cache
    - e.g. explicit blocking to shared memory on NVIDIA GPUs
- **CCE-specific tuning:**
  - can also use !dir$ directives to adjust loop scheduling
    - e.g. concurrent, blockable
  - see man intro_directives (with PrgEnv-cray loaded) for details

More OpenACC directives

- **!$acc update [host|device]**
  - Copy specified arrays (slices) within data region
  - Useful if you only need to send a small subset of data to/from GPU
    - e.g. halo exchange for domain-decomposed parallel code
    - or sending a few array elements to the CPU for printing/debugging
  - Remember slicing syntax differs between Fortran and C/C++
  - The array sections should be contiguous
    - a(2:N-1,1:N) is not OK, but a(1:N,2:N-1) is OK
- **!$acc declare**
  - Makes a variable resident in accelerator memory
    - persists for the duration of the implicit data region
- **Other directives**
  - We'll cover these in detail later:
    - !$acc cache
    - async clause and !$acc wait
    - !$acc host_data

parallel vs. kernels

- **parallel and kernels regions look very similar**
  - both define a region to be accelerated
    - different heritage; different levels of obligation for the compiler
  - parallel
    - prescriptive (like OpenMP programming model)
    - uses a single accelerator kernel to accelerate region
    - compiler will accelerate region (even if this leads to incorrect results)
  - kernels
    - descriptive (like PGI Accelerator programming model)
    - uses one or more accelerator kernels to accelerate region
    - compiler may accelerate region (if decides loop iterations are independent)
  - For more info: http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

- **Which to use (my opinion)**
  - parallel (or parallel loop) offers greater control
    - fits better with the OpenMP model
  - kernels (or kernels loop) better for initially exploring parallelism
    - not knowing if loopnest is accelerated could be a problem

parallel loop vs. parallel and loop

- **parallel region can span multiple code blocks**
  - i.e. sections of serial code statements and/or loopnests
  - loopnests in parallel region are not automatically partitioned
    - need to explicitly use loop directive for this to happen
  - scalar code (serial code, loopnests without loop directive)
    - executed redundantly, i.e. identically by every thread
      - or maybe just by one thread per block (its implementation dependent)
  - There is no synchronisation between redundant code or kernels
    - offers potential for overlap of execution on GPU
    - also offers potential (and likelihood) of race conditions and incorrect code
  - There is no mechanism for a barrier inside a parallel region
    - after all, CUDA offers no barrier on GPU across threadblocks
    - to effect a barrier, end the parallel region and start a new one
      - also use wait directive outside parallel region for extra safety

parallel loop vs. parallel and loop

- **My advice: don't...**
  - GPU threads are very lightweight (unlike OpenMP)
    - so don't worry about having extra parallel regions
  - explicit use of async clause may achieve same results
    - as using one parallel region
    - but with greater code clarity and better control over overlap

- **... but if you feel you must**
  - begin with composite parallel loop and get correct code
    - separate directives with care only as a later performance tuning
      - when you are sure the kernels are independent and no race conditions
    - this is similar to using OpenMP on the CPU
      - if you have multiple do/for directives inside omp parallel region
      - only introduce nowait clause when you are sure the code is working
      - and watch out for race conditions

parallel gotchas

- **No loop directive**
  - The code will (or may) run redundantly
    - Every thread does every loop iteration
    - Not usually what we want

```
!$acc parallel
  DO i = 1,N
   a(i) = b(i) + c(i)
  ENDDO
!$acc end parallel
```

- **Serial code in parallel region**
  - avoids copyin(t), but a good idea?
  - No! Every thread sets t=0
  - asynchronicity: no guarantee this finishes before loop kernel starts
  - race condition, unstable answers.

```
!$acc parallel
  t = 0
!$acc loop reduction(+:t)
  DO i = 1,N
   t = t + a(i)
  ENDDO
!$acc end parallel
```

- **Multiple kernels**
  - Again, potential race condition
  - Treat OpenACC "end loop" like OpenMP "enddo nowait"

```
!$acc parallel
!$acc loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc loop
  DO i = 1,N
   a(i) = a(i) + 1
  ENDDO
!$acc end parallel
```

parallel loop vs. parallel and loop

- **My advice: when you actually might want to**
  - You *might* split the directive if:
    - you have a single loopnest, and
    - you need explicit control over the loop scheduling
    - you do this with multiple loop directives inside parallel region
      - or you could use parallel loop for the outermost loop, and loop for the others
- **But beware of reduction variables**
  - With separate loop directives, you need a reduction clause on every loop directive that includes a reduction:

```
t = 0
!$acc parallel loop &
!$acc    reduction(+:t)

DO j = 1,N

  DO i = 1,N
    t = t + a(i,j)
  ENDDO
ENDDO
!$acc end parallel loop
```

Correct!

```
t = 0
!$acc parallel &
!$acc    reduction(+:t)
!$acc loop
DO j = 1,N
!$acc loop
  DO i = 1,N
    t = t + a(i,j)
  ENDDO
ENDDO
!$acc end parallel
```

Wrong!

```
t = 0
!$acc parallel

!$acc loop reduction(+:t)
DO j = 1,N
!$acc loop
  DO i = 1,N
    t = t + a(i,j)
  ENDDO
ENDDO
!$acc end parallel
```

Wrong!

```
t = 0
!$acc parallel

!$acc loop reduction(+:t)
DO j = 1,N
!$acc loop reduction(+:t)
  DO i = 1,N
    t = t + a(i,j)
  ENDDO
ENDDO
!$acc end parallel
```

Correct!

The OpenACC runtime API

- **Directives are comments in the code**
  - automatically ignored by non-accelerating compiler
- **OpenACC also offers a runtime API**
  - set of library calls, names starting acc_
    - set, get and control accelerator properties
    - offer finer-grained control of asynchronicity
  - OpenACC specific
    - will need pre-processing away for CPU execution
    - #ifdef _OPENACC
- **CCE offers an extended runtime API**
  - set of library calls, names starting with cray_acc_
    - will need preprocessing away if not using OpenACC with CCE
    - #if defined(_OPENACC) && PE_ENV==CRAY
- **Advice: you do not need the API for most codes.**
  - Start without it, only introduce it where it is really needed.
  - I almost never use it; we'll talk about it later.

Sources of further information

- **OpenACC standard web page:**
  - OpenACC.org
    - documents: full standard and quick reference guide PDFs
    - links to other documents, tutorials etc.
- **Discussion lists:**
  - Cray users: openacc-users@cray.com
    - automatic subscription if you have a raven account
  - OpenACC forum: openacc.org/forum

- **CCE man pages (with PrgEnv-cray loaded):**
  - programming model and Cray extensions: intro_openacc
  - examples of use: openacc.examples
  - also compiler-specific man pages: crayftn, craycc, crayCC

- **CrayPAT man pages (with perftools loaded):**
  - intro_craypat, pat_build, pat_report
    - also command: pat_help
  - accpc (for accelerator performance counters)

More sources of further information

- **PGI Accelerator, including PGI Insider newsletter articles**
  - http://www.pgroup.com/lit/articles/insider/v4n1a1.htm (overview of support)
  - http://www.pgroup.com/lit/articles/insider/v4n2a1.htm (parallel vs. kernels)
- **CAPS**
- **Recent ORNL workshop**
  - http://www.olcf.ornl.gov/event/cray-technical-workshop-on-xk6-programming/
- **Recent training courses**
  - PRACE, Cray:
    - http://www.epcc.ed.ac.uk/training-education/course-programme/gpu-programming-workshop/
  - SC12 tutorials:
    - Cray:
      - Productive, Portable Performance on Accelerators Using OpenACC Compilers and Tools
    - PGI:
      - Introduction to GPU Computing with OpenACC
      - Advanced GPU Computing with OpenACC