An introduction to OpenACC

Alistair Hart (Cray),
presented by Will Sawyer (CSCS)

Refresher: what is important for GPUs?

- You need a lot of parallel tasks (i.e. loop iterations) to keep GPU busy
  - Each parallel task maps to a thread in a threadblock
  - You need a lot of threadblocks per streaming multiprocessor (SM) to hide memory latency
  - Not just 2688 parallel tasks, but 10^4 to 10^6 or more
  - This is most-likely in a loop-based code, treating iterations as tasks
    - OpenACC is particularly targeted at loop-based codes

- Your inner loop must vectorise (at least with vector length of 32)
  - So we can use all 32 threads in a warp with shared instruction stream
  - Branches in inner loop are allowed, but not too many

- Memory should be accessed in the correct order
  - Global memory access is done with (sequential) vector loads
  - For good performance, want as few of these as possible
  - so all the threads in warp should collectively load a contiguous block of memory at the same point in the instruction stream
  - This is known as "coalesced memory access"
  - So vectorised loop index should be fastest-moving index of each array

What does this mean for the programmer?

- No internal mechanism for synchronising between threadblocks
  - Synchronisation must be handled by host
    - So reduction operations are more complicated
    - even though all threadblocks share same global memory
  - Fortunately launching kernels is cheap
    - GPU threadteams are "lightweight"

- Data transfers between CPU and GPU are very expensive
  - You need to concentrate on "data locality" and avoid "data sloshing"
  - Keeping data in the right place for as long as it is needed is crucial
  - You should port as much of the application as possible
    - This probably means porting more than you expected

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

Directive-based programming

Directives provide a high-level alternative

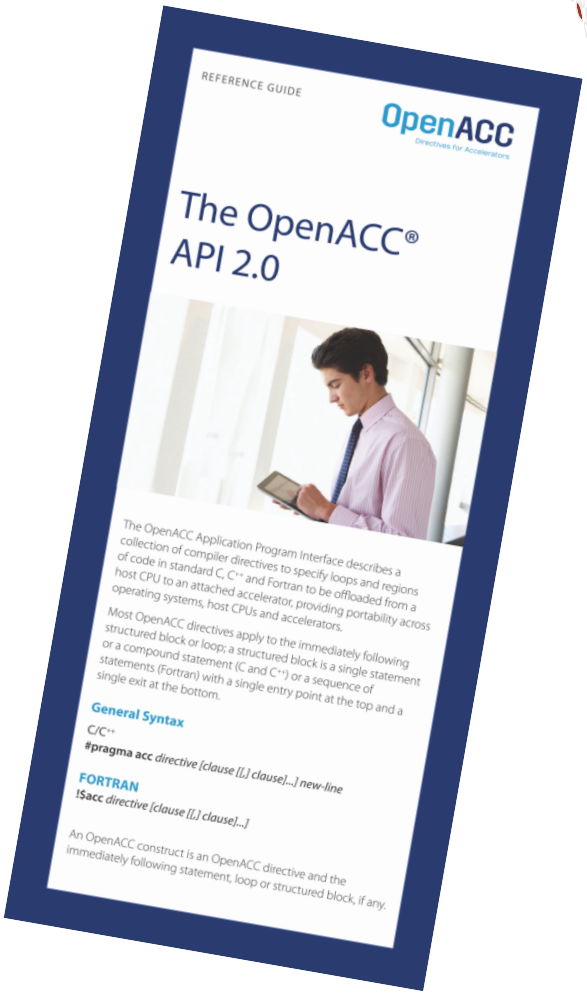
+ **Based on original source code (Fortran, C, C++)**
  + Easier to maintain/port/extend code
  + Users with OpenMP experience find it a familiar programming model
  + Compiler handles repetitive coding (cudaMalloc, cudaMemcpy...)
  + Compiler handles default scheduling; user tunes only where needed

– **Possible performance sacrifice**
  – Important to quantify this
  – Can then tune the compiler
  – Small performance sacrifice is acceptable
    – trading-off portability and productivity against this
    – after all, who hand-codes 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: version 12.6 onwards
  - CAPS: Full support in v1.3
  - gcc:work started in late 2013, aiming for 4.9
  - Various other compilers in development

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

Strategic risk factors of OpenACC

- **Will there be machines to run my OpenACC code on?**
  - Now? Lots of Nvidia GPU accelerated systems
    - Cray XC30s and XK7s, plus other vendors (OpenACC is multi-vendor)
  - Future? OpenACC can be targeted at other accelerators
    - PGI and CAPS already target Intel Xeon Phi, AMD GPUs
  - Plus you can always run on CPUs using same codebase

- **Will OpenACC continue?**
  - Support? Cray, PGI, CAPS committed to support. Now gcc as well.
    - Lots of big customer pressure to continue to run OpenACC
  - Develop? OpenACC committee now 18 partners
    - v2.0 finalised in 2013, now working on next version (2.1 or 3.0)

- **Will OpenACC be superseded by something else?**
  - Auto-accelerating compilers? Yes, please! But never managed before
    - Data locality adds to the challenge
  - OpenMP accelerator directives? Immature at the moment
    - OpenACC work not wasted: thinking takes more time than coding
    - Very similar programming model; can transition when these release if wish
    - Cray (co-chair), PGI very active in OpenMP accelerator subcommittee

OpenACC suitability

- **Will my code accelerate well with OpenACC?**
  - Computation should be based around loopnests processing arrays
    - Loopnests should have defined tripcounts (either at compile- or run-time)
      - while loops will not be easy to port with OpenACC
      - because they are hard to execute on a GPU
    - Data structures should be simple arrays
      - derived types, pointer arrays, linked lists etc. may stretch compiler capabilities
  - The loopnests should have a large total number of iterations
    - at least measured in the thousands
      - even more is better; less will execute, but with very poor efficiency
  - The loops should span as much code as possible
    - maybe with some loops very high up the callchain
  - The loopnest kernels should not be too branched
    - one or two nested IF-statements is fine
    - too many will lead to slow execution on many accelerators
  - The code can be task-based
    - but each task should contain a suitable loopnest

So...

- **GPUs can give very good performance**
  - but you need to be aware of the underlying architecture
  - porting a real application to GPU(s) requires some hard work
    - Amdahl says you need to port a lot of the profile to see a speed-up
      - bad news: to see 10x speedup, need to port at least 90% of the application profile
      - good news: if profile very peaked, 90% of time may be spent in, say, 40% of code
    - even before you worry about the costs of data transfers

- **A good programming model and environment**
  - helps bridges the gap between peak and achievable performance

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 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 threads of GPU
    - OpenACC will "partition" (workshare) more than one loop in a loopnest
    - compare: OpenMP only partitions the outer loop
  - Caching (e.g. explicit use GPU shared memory for reused data)
    - automatic caching can be important
  - Tune 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 kernel from a loopnest
      - 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

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 "like a broken pencil"**
  - pointless (that is, redundant)

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 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)

- **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 or 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 nested parallelism

Reduction variables

- **Reduction variables are a special case of private 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 private 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
  - if you are translating an OpenMP code to OpenACC

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)
    - Note 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 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

#pragma acc exit data

Do you have any questions?