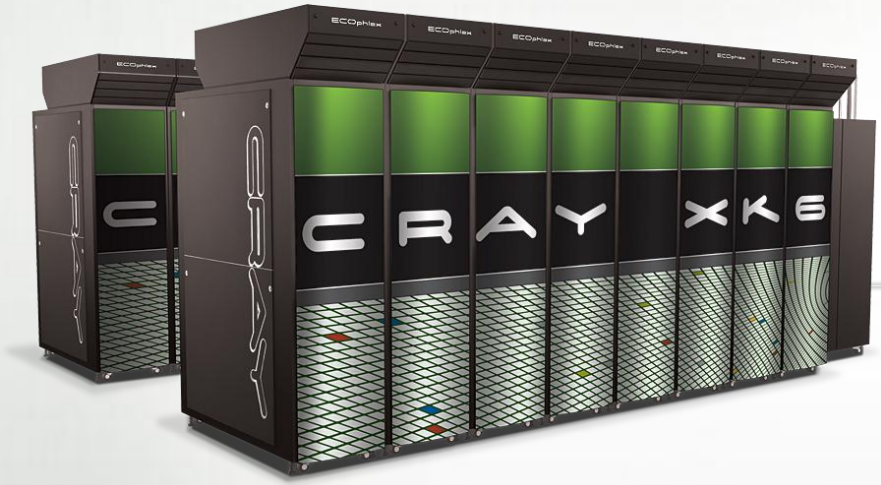
Programming GPU Devices using OpenACC directives on the Cray XK6 platform



Alistair Hart[†],
Roberto Ansaloni.

([†]Cray Exascale Research Initiative Europe)

CSCS – Tuesday 6.Mar.12

ahart@cray.com

Contents

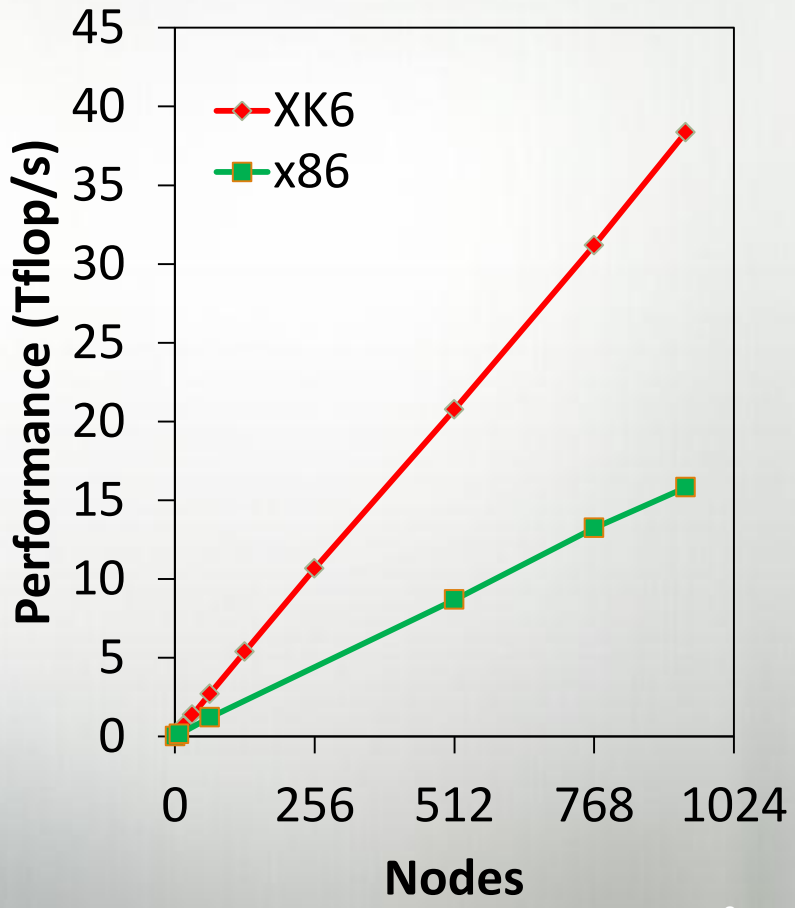
- Accelerator directives
  - Why do we need them?
  - What do they look like?
    - OpenACC now, OpenMP in the future
  - How do we use them?
  - Support status in CCE v8.0
- Use Cases:
  - How do we port a full application?
  - How do they perform?
  - Case studies in directive-based optimisation on GPU
    - S3D: performance
    - Himeno: porting a parallel benchmark
    - MultiGrid: now it's your turn
- Running an existing CUDA or OpenCL application
- Overview of MultiGrid tutorial example

Accelerator programming

- Why do we need a new GPU programming model?
- Aren't there enough ways already?
  - CUDA (incl. NVIDIA CUDA-C & PGI CUDA-Fortran)
  - OpenCL
  - Stream
  - hiCUDA ...
- 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

CUDA on Cray XK6

- If you work hard, you can get good parallel performance
- Ludwig Lattice Boltzmann code rewritten in CUDA
  - Reordered all the data structures (structs of arrays)
  - Pack halos on the GPU
  - Streams to overlap compute, PCIe comms, MPI halo swaps
- 10 cabinets of Cray XK6
  - 936 GPUs (nodes)
- Only 4% deviation from perfect weak scaling between 8 and 936 GPUs.
- Application sustaining 40+ Tflop/s

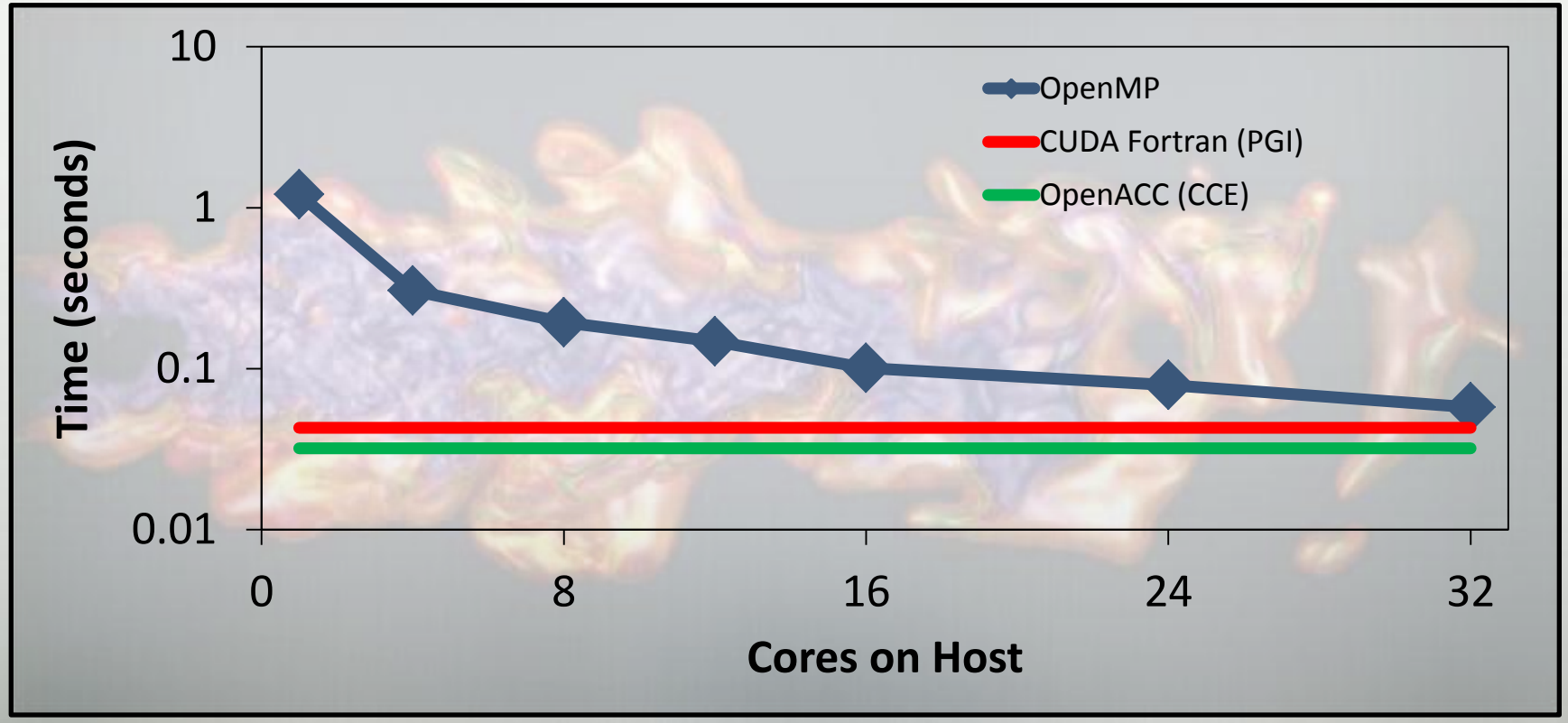


Directive-based programming

- Most scientific applications will not have this level of developer support (Ludwig was special research case)
- Directives provide high-level approach
  - + Based on original source code (e.g. Fortran, C, C++)
    - + Easier to maintain/port/extend code
    - + Users with (for instance) OpenMP experience find it a familiar programming model
    - + Compiler handles repetitive boilerplate code (cudaMalloc, cudaMemcpy...)
    - + Compiler handles default scheduling; user can step in with clauses where needed
  - – Possible performance sacrifice
    - – Important to quantify this
    - – Can then tune the compiler
    - – Small performance sacrifice is an acceptable trade-off for portability and productivity
      - – Who handcodes in assembler these days?
- Two relevant performance comparisons:
  - How does the performance compare to CUDA?
  - Can I justify buying a GPU instead of another CPU?

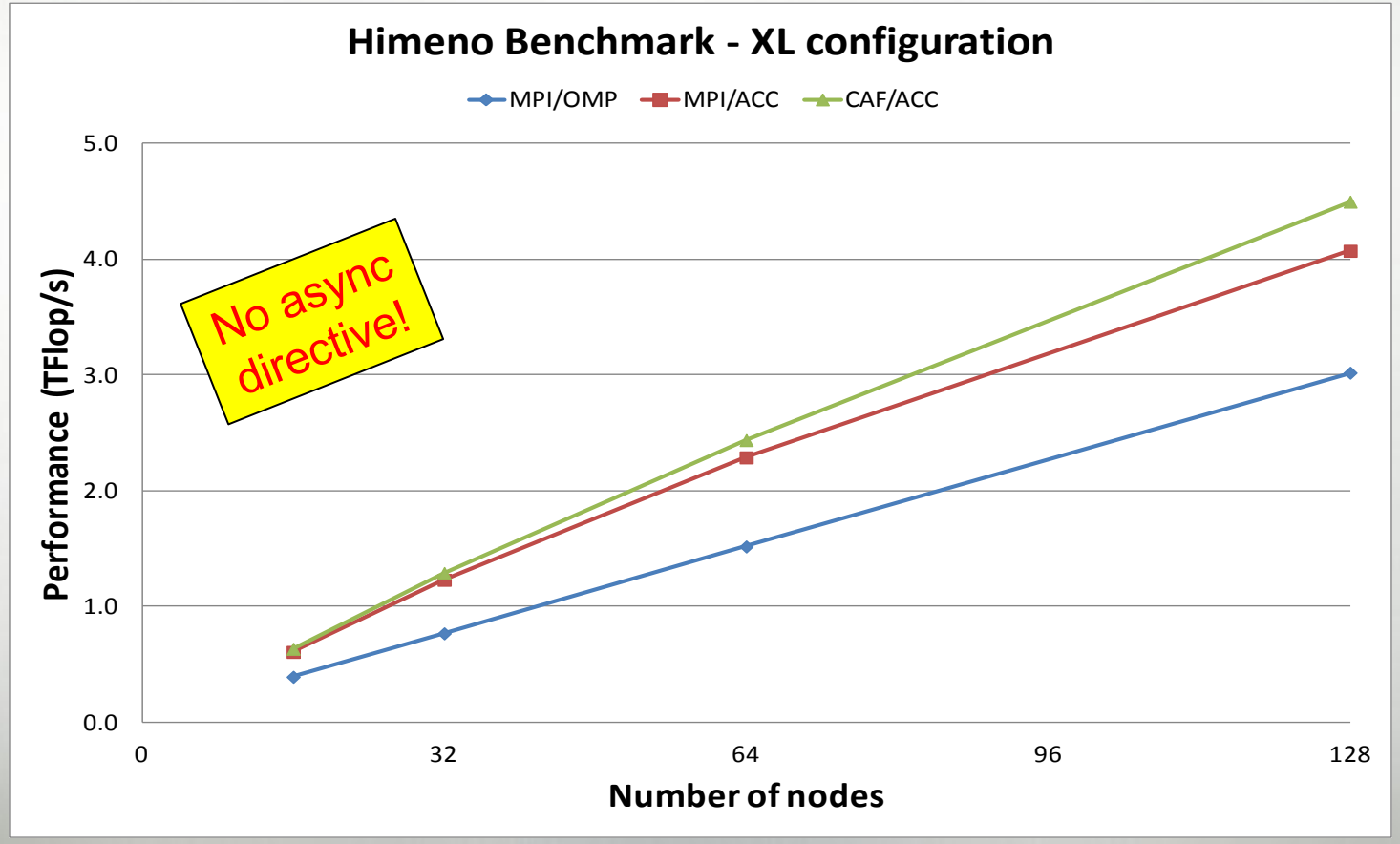
Performance compared to CUDA

- Is there a performance gap relative to explicit low-level programming model? Typically 10-15%, sometimes none.
- Is the performance gap acceptable? Yes.
  - e.g. S3D comp_heat kernel (ORNL application readiness):



Node-for-node performance comparison

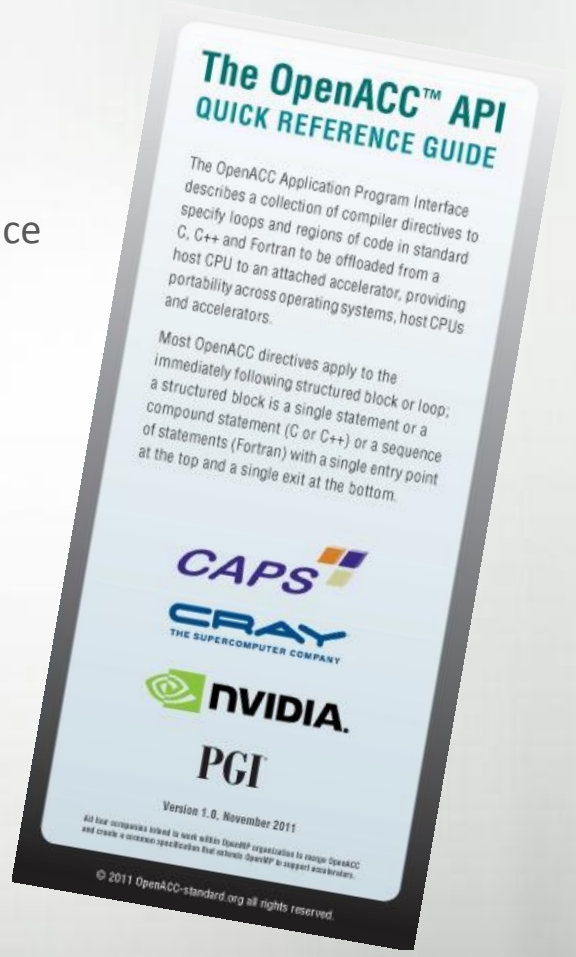
- Does accelerated parallel application performance justify buying a GPU (Cray XK6) rather than another CPU (Cray XE6)?
  - For many codes, yes.



OpenACC®
DIRECTIVES FOR ACCELERATORS

CRAY THE SUPERCOMPUTER COMPANY

- 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 www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 1.0 (November 2011)
- Compiler support:
  - Cray CCE: partial now, complete in 2012
  - PGI Accelerator: released product in 2012
  - CAPS: released product in Q1 2012



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
- Subcommittee of OpenMP ARB
  - includes most major vendors + others (e.g. EPCC)
  - co-chaired by Cray (James Beyer)
  - aiming for OpenMP 4 (2012?)
- Targets Fortran, C, C++
- Current version: draft
- Cray compiler provides reference implementation for ARB
  - Of draft standard at present (CCE 8.0)
  - Will track the standard as it evolves
- Converting from OpenACC to OpenMP will be straightforward

OpenACC Execution model

- Host-directed execution with attached GPU
  - 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

- 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 also supresses compilation
  - Sentinel: !$acc
  - Fortran:
    - Usually paired with !$acc end *
  - C/C++:
    - Structured block {...} avoids need for end directives
  - Continuation to extra lines allowed

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

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

- CPP macro defined to allow extra conditional compilation
  - E.g. around calls to runtime API functions
    - _OPENACC == yyyymm (currently 201111)

A first example

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

write-only

read-only

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
  - Loop schedule: spreading loop iterations over PEs of GPU

Parallelism	**NVIDIA GPU**	**SMT node** ***(not supported!)***
gang:	a threadblock	CPU
worker:	warp (32 threads)	CPU core
vector:	SIMT group of threads	SIMD instructions (SSE, AVX)

  - Caching (explicitly use GPU shared memory for reused data)
    - automatic caching (e.g. NVIDIA Fermi) important
  - Tune default behaviour with optional clauses on directives

A first 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
- 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>
!$acc data copy(a)
!$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
    - Compiler will automatically inline (maybe need `-Oipafrom` or use program library)
- The 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 (can be shortened to pcopy)
- Original calltree structure of program can be preserved

Clauses for !$acc parallel loop

- Data clauses:
  - 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 Fortran ":" notation)
  - create
    - No copyin/out – useful for shared temporary arrays in loopnests
  - private: scalars private by default
  - present, present_or_copy*: described previously
- Tuning clauses:
  - !$acc loop [gang] [worker] [vector]
    - Targets specific loop (or loops using collapse clause) at specific level of hardware
  - num_gang, num_workers, vector_length
    - Tunes the amount of parallelism used (threadblocks, threads/block...)
  - seq: loop executed sequentially
  - independent: compiler hint (also use CCE !dir$ directives)

More OpenACC directives

- Other !$acc parallel loop clauses:
  - if(logical)
    - Executes on GPU if .TRUE. at runtime, otherwise on CPU
  - reduction: as in OpenMP
  - cache: specified data held in software-managed data cache
    - e.g. explicit blocking to shared memory on NVIDIA GPUs
- !$acc update [host|device]
  - Copy specified arrays (slices) within data region
- async[(handle)] clause for parallel, update directives
  - Launch accelerator region/data transfer asynchronously: allows CPU/GPU overlap
  - Operations with same handle will execute sequentially (as in CUDA streams)
  - !$acc wait[(handles)]: waits for completion
  - Runtime library functions can also be used to test/wait for completion
  - Will be supported in CCE v8.1

Interoperability with CUDA

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copy(a)
! <Populate a(:) on device
!  as before>
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  <stuff>
END PROGRAM main
```

```
__global__ void dbl_knl(int *c) {
  int i = \
       blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- host_data region exposes accelerator memory address on host
  - nested inside data region
- Call CUDA-C wrapper (compiled with nvcc; linked with CCE)
  - must include cudaThreadSynchronize()
  - Before: so asynchronous accelerator kernels definitely finished
  - After: so CUDA kernel definitely finished
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

OpenACC support status in CCE v8.0

- parallel
  - Supported: num_coarse, num_fine, vector_length, all data clauses
  - Unsupported: async, if
- kernel
  - Unsupported
- loop
  - Supported: collapse, coarse, fine, vector, reduction, private, seq
  - Unsupported: independent
- data
  - Supported: all data clauses
  - Unsupported: async, if
- host_data
  - Supported
- acc_update
  - Supported: host, device
  - Unsupported: async, if
- Unsupported
  - Directives : wait, cache, declare
  - All runtime routines

A porting strategy

- Preparation: add checksum(s) and high-res timer to code
  - Check for correctness very frequently
  - Profile code on the host
    - Use representative-sized problem, map calltree,
    - Ideally resolve profile by loopnest and measure typical loop iteration counts
- First optimise the data movements
  - Start in subprograms at bottom of callchain
    - Accelerate individual loopnests using parallel regions
      - Concentrate initially on most computationally expensive
    - Add data regions in subprograms
      - Minimise data movements, use create clause where possible
      - May need to accelerate insignificant loopnests to avoid data copies
  - Use available feedback to understand data movement
    - Compiler messages: -ra for CCE
    - Runtime commentary: export CRAY_ACC_DEBUG=[1,2,3] for CCE
    - NVIDIA compute profiler: export COMPUTE_PROFILE=1
    - CrayPAT performance measurement and analysis tool (Cray PE only)
- Code is probably going quite slowly at this point

A porting strategy (2)

- Move progressively up callchain, adding data regions
  - Aim to further reduce data movements
    - No problem nesting data regions: use present clause on inner ones
    - May need to port insignificant subprograms to avoid data transfers
    - Use update for essential data transfers (e.g. data for halo swaps)
- Now optimise kernel performance (often trial and error)
  - Perfect loop nests schedule better than imperfect ones
    - e.g. Remove temporary arrays by manually inlining (eliminate array b)
    - Or manually privatise arrays and break loopnest (make b(i,j))

```
DO j = 1,N
 DO i = 0,M+1
  b(i) = a(i,j+1) + a(i,j-1)
 ENDDO
 DO i = 1,M
  c(i,j) = b(i+1) + b(i-1)
 ENDDO
ENDDO
```

```
DO j = 1,N
 DO i = 1,M
  c(i,j) = a(i+1,j+1) + a(i+1,j-1) &
         + a(i-1,j+1) + a(i-1,j-1)
 ENDDO
ENDDO
```

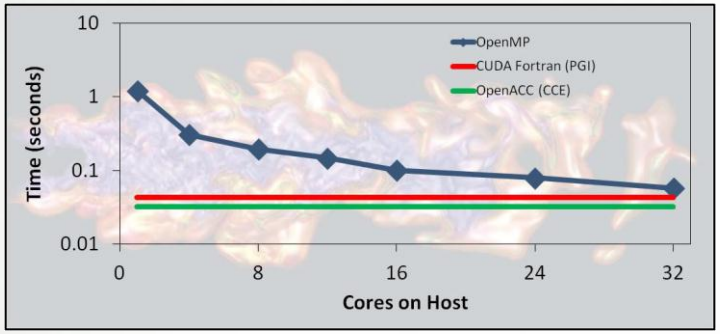
```
DO j = 1,N
 DO i = 0,M+1
  b(i,j) = a(i,j+1) + a(i,j-1)
 ENDDO
ENDDO
DO j = 1,N
 DO i = 1,M
  c(i,j) = b(i+1) + b(i-1)
 ENDDO
ENDDO
```

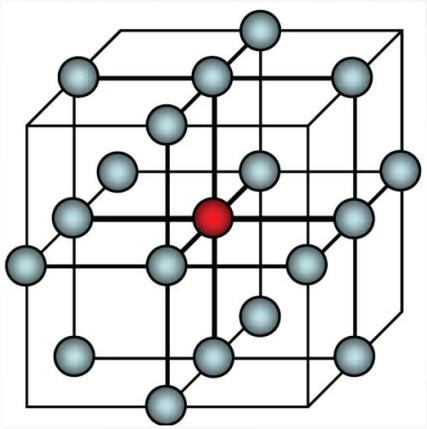
A porting strategy (3)

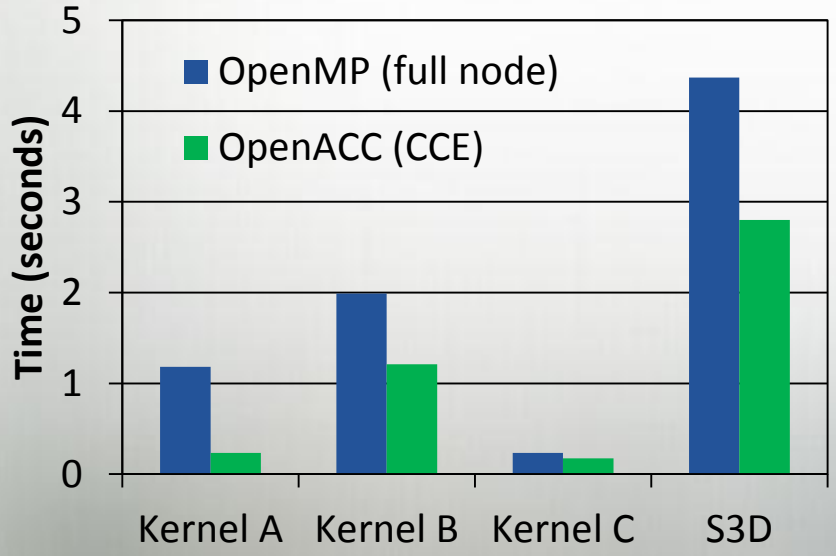
- Now look at tweaking the loop scheduling
  - Quick wins
    - Optimise loop scheduling
      - Make sure the right loops are vectorised (for coalesced memory loads)
        - And that they are vectorisable
      - Choose number of workers per gang (threads/block)
        - This number will vary by kernel and by problem size
        - Collapsing or blocking of loops may help (though compilers already do that)
    - See if caching can be used to reduce data loads from device memory
  - Longer term: can loops be migrated up the callchain?
    - E.g. Loop over sites, or blocks of sites (“blocking for cache”)
    - If so, parallelise (gangs) over these
- Consider overlap of compute and communications using async
  - Don’t do this until everything working
  - May require application restructuring

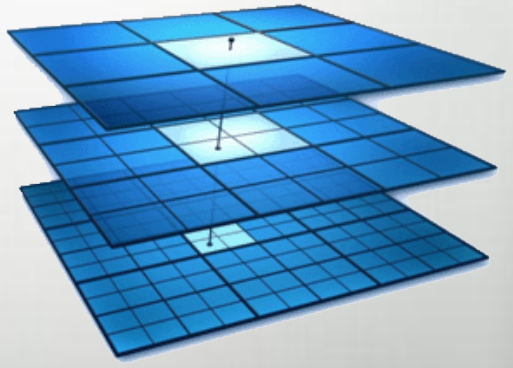
Three example applications

1. S3D turbulent combustion code
2. Himeno
3. MultiGrid code (NAS & SPEC benchmarks)

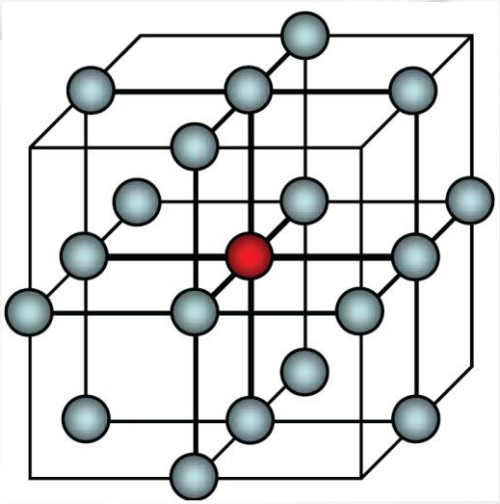




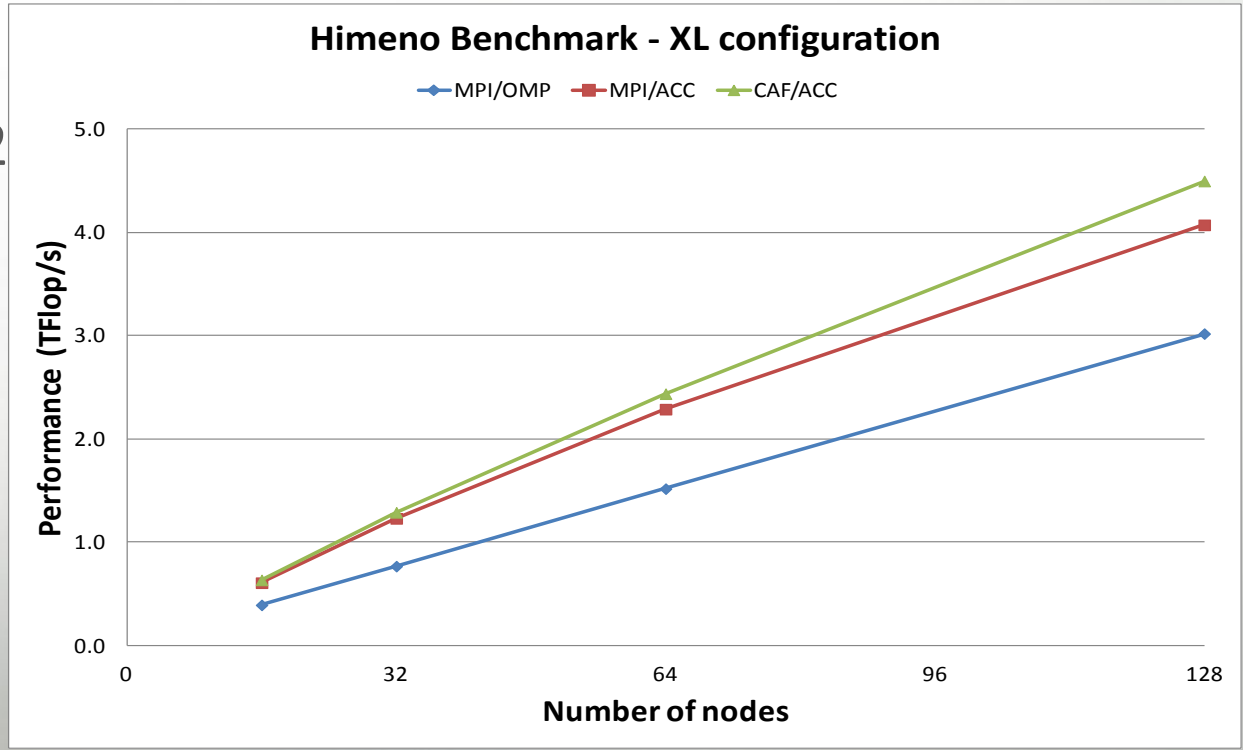




Example: The Himeno Benchmark



- Parallel 3D Poisson equation solver
  - 19-point stencil
- MPI or CAF and/or OpenMP
  - available from here
  - ~600 lines of Fortran
  - Fully ported to accelerator using 27 directive pairs
- XL configuration:
  - 1024 x 512 x 512
  - Strong scaling
- More kernel tuning
- No use of async yet



The Jacobi computational kernel

- The stencil is applied to pressure array p
- Updated pressure values are saved to temporary array wrk2
- Control value wgosa is computed
- In the benchmark this kernel is iterated a fixed number of times (nn)

```
DO K=2,kmax-1
 DO J=2,jmax-1
  DO I=2,imax-1
   S0=a(I,J,K,1)*p(I+1,J, K )
     +a(I,J,K,2)*p(I, J+1,K ) &
     +a(I,J,K,3)*p(I, J, K+1) &
     +b(I,J,K,1)*(p(I+1,J+1,K )-p(I+1,J-1,K ) &
                 -p(I-1,J+1,K )+p(I-1,J-1,K )) &
     +b(I,J,K,2)*(p(I, J+1,K+1)-p(I, J-1,K+1) &
                 -p(I, J+1,K-1)+p(I, J-1,K-1)) &
     +b(I,J,K,3)*(p(I+1,J, K+1)-p(I-1,J, K+1) &
                 -p(I+1,J, K-1)+p(I-1,J, K-1)) &
     +c(I,J,K,1)*p(I-1,J, K ) &
     +c(I,J,K,2)*p(I, J-1,K ) &
     +c(I,J,K,3)*p(I, J, K-1) &
     + wrk1(I,J,K)

   SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
   WGOSA=WGOSA+SS*SS
   wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
  ENDDO
 ENDDO
ENDDO
```

fwd n.n.

n.n.n.

bkwd n.n.

Distributed implementation

- Outer loop executed fixed number of times
- Jacobi kernel is executed and new pressure array wrk2 and control value wgosa computed
- array is updated with the new pressure values
- halo region values are exchanged between neighbour PEs
- Send/receive buffers are used
- The maximum control value is computed with an ALLREDUCE operation across all the PEs

```
DO loop = 1, nn
   compute Jacobi kernel → wrk2,wgosa

   copy back wrk2 into p

   pack halo from p into send buffers

   exchange halos with neighbour PEs

   unpack halo into p from recv buffers

   Allreduce to sum wgosa across PEs
ENDDO
```

Porting Himeno to the Cray XK6

- Several versions tested, with communication implemented in MPI or Fortran coarrays
- GPU version using OpenACC ccelerator directives
- Comparing Cray XK6 timings with best Cray XE6 results (hybrid MPI/OpenMP)
- Arrays reside permanently on the GPU memory
- Data transfers between host and GPU are:
  - Communication buffers for the halo exchange
  - Control value

Allocating arrays on the GPU

- Arrays are allocated on the GPU memory in the main program with the `data` directive
- In the subroutines the `data` directive is replicated with the `present` clause, to use the data already present in the GPU memory and avoid extra allocations
- Since `present` clause is used, no `copy*` clauses are used, and data transfers to/from host are implemented by `update` directives

```
PROGRAM himenobmtxp
...
!$acc data create          &
!$acc&   (p,a,b,c,wrk1,wrk2,bnd,      &
!$acc&   sendbuffx_up,sendbuffx_dn, &
!$acc&   sendbuffy_up,sendbuffy_dn, &
!$acc&   sendbuffz_up,sendbuffz_dn)
...
!$acc end data

SUBROUTINE jacobi(nn,gosa)
!$acc data present          &
!$acc&   (p,a,b,c,wrk1,wrk2,bnd,      &
!$acc&   sendbuffx_up,sendbuffx_dn, &
!$acc&   sendbuffy_up,sendbuffy_dn, &
!$acc&   sendbuffz_up,sendbuffz_dn)
```

Jacobi kernel on the GPU

- The GPU kernel for the main loop is created with the `parallel loop` directive
- The scoping of the main variables is specified earlier with the `data` directive - no need to replicated it in here
- wgosa is computed by specifying the `reduction` clause, as in a standard OpenMP parallel loop
- `vector_length` clause is used to indicate the number of threads within a threadblock (compiler default 128)

```
DO loop=1,nn
  gosa = 0
  wgosa = 0
!$acc parallel loop                &
!$acc&  private(s0,ss)               &
!$acc&  reduction(+:wgosa)           &
!$acc&  vector_length(256)
  DO K=2,kmax-1
    DO J=2,jmax-1
      DO I=2,imax-1
        S0=a(I,J,K,1)*p(I+1,J, K ) &
        ...
        wgosa = wgosa + SS*SS
      ENDDO
    ENDDO
  ENDDO
```

Halo region buffers

- Halo values are extracted from the wrk2 array and packed into the send buffers, on the GPU
- A global `parallel` region is specified and buffers in the X, Y, and Z directions are packed within `loop` blocks
- The send buffers are copied to host memory with `update`
- In the same way, after the halo exchange, the recv buffers are transferred to the GPU memory and used to update the array p
- N.B. Currently it's not possible to include non-contiguous array sections in `update`
  - buffers are necessary

```
!$acc parallel
!$acc loop
DO j = 2,jmax-1
  DO i = 2,imax-1
    sendbuffz_dn(i,j)= wrk2(i,j,2)
    sendbuffz_up(i,j)= wrk2(i,j,kmax-1)
  ENDDO
ENDDO
!$acc end loop
 ...
!$acc loop
!$acc end loop
!$acc end parallel

!$acc update &
!$acc&  host(sendbuffz_dn,sendbuffz_up)
```

Coarray implementation

- Coarrays are used to perform the halo exchange
- Non-blocking communication needs `pgas defer_sync` directive
- Programmer now responsible for data synchronization
- By deferring sync point, network comms can be overlapped with CPU or GPU activity
- Updating p from wrk2 (on GPU) overlapped with halo exchange
- N.B. no `sync all`: CAF intrinsic COSUM has loose synchronisation (so do need `sync memory` first).

```
!dir$ pgas defer_sync
recvbuffz_up(:,:)[myx,myy,myz-1] = &
    sendbuffz_dn(:,:)
 ...
!$acc parallel loop
DO k = 2,kmax-1
  DO j = 2,jmax-1
    DO i = 2,imax-1
      p(i,j,k) = wrk2(i,j,k)
    ENDDO
  ENDDO
ENDDO
!$acc end parallel loop
sync memory
gosa = COSUM(wgosa)
!$acc update &
!$omp& acc(recvbuffz_dn,recvbuffz_up)
```

Coarray implementation

- Coarrays are used to perform the halo exchange
- Non-blocking communication needs `pgas defer_sync` directive
- Programmer now responsible for data synchronization
- By deferring sync point, the comms can be overlapped with CPU or GPU activity
- Updating p from wrk2 (on GPU) overlapped with halo exchange
- N.B. no `sync all`: CAF intrinsic `COSUM` has loose synchronisation (so do need `sync memory` first).

Compiler does not currently support using coarrays in an accelerator region, so this does not work!

You need to make a local copy of the coarray buffers to non-coarray buffers and then transfer them to GPU memory.

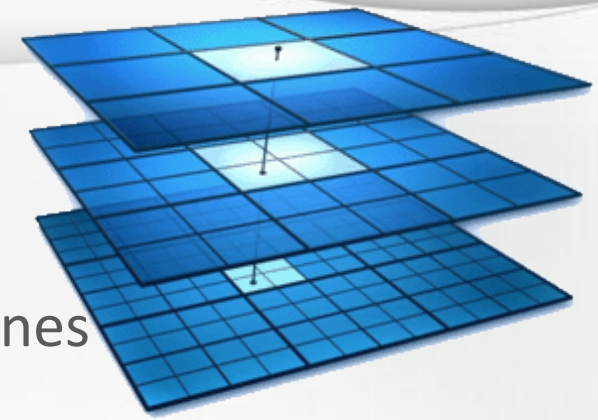
This affects the performance, by increasing the host CPU time.

```
!$omp acc_region_loop
DO k = 2,kmax-1
DO j = 2,jmax-1
DO i = 2,imax-1
p(i,j,k) = wrk2(i,j,k)
ENDDO
ENDDO
ENDDO
!$omp end acc_region_loop
sync memory
gosa = COSUM(wgosa)
!$omp acc_update &
!$omp& acc(recvbuffz_dn,recvbuffz_up)
```

OpenMP for Accelerator GPU version

- Total number of lines in the original Himeno MPI-Fortran code: 629
- Total number lines in the modified version with coarrays and accelerator directives: 554
  - don't need MPI_CART_CREATE and the like
- Total number of accelerator directives: 27
  - plus 18 "end" directives

Example: MultiGrid benchmark



- NAS Parallel Benchmarks, also SPEC suite
- MG (multigrid) solves Laplacian on 3D grid
  - 1500 lines of Fortran or C, many subroutines
  - Three main hotspots:
    - **resid** (50% of runtime), **psinv** (25%), **rprj3** (9%)
  - Data arrays passed to/from subroutines at every iteration
- GPU (just less than) 2x faster than CPU (16 cores)
  - Fully accelerated using 25 directive pairs (present essential)
- You will look at this code in the tutorial this afternoon
- MPI-parallel version also ported using OpenACC
- Further optimisations coming
  - Further use of shared memory
  - async clause support coming
    - CCE already launches kernels and data transfers asynchronously

Running existing CUDA applications

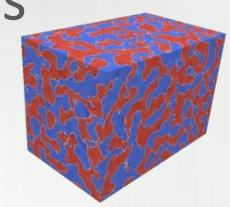
- CUDA codes can be compiled and run as usual on the Cray XK6
- Ludwig parallel code was run across 936 GPUs (10 cabinets)
- Compilation:
  - `module load craype-accel-nvidia20`
  - Main CPU code compiled with PrgEnv "cc" wrapper
    - either PrgEnv-gnu for gcc; or PrgEnv-cray for craycc
  - GPU CUDA-C kernels compiled with nvcc
    - `nvcc -O3 -arch=sm_20`
  - PrgEnv "cc" wrapper used for linking
    - Only GPU flag needed: `-lcudart`
      - e.g. no CUDA `-L` flags needed (added in cc wrapper)
- Submission:
  - submit job as usual (SLURM, aprun):
    - Use 1 MPI rank per node
  - NVIDIA drivers for Cray XK6 optimise GPU/CPU/Gemini pipeline.

Running existing OpenCL applications

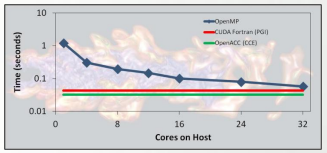
- Compilation:
  - `module load craype-accel-nvidia20`
  - Main CPU code compiled with PrgEnv "cc" wrapper
    - either PrgEnv-gnu for gcc; or PrgEnv-cray for craycc
  - GPU OpenCL kernels compiled with nvcc
    - `nvcc -O3 -arch=sm_20`
  - PrgEnv "cc" wrapper used for linking
    - Only GPU flag needed: `-lOpenCL`
- Alternatively:
  - Use PrgEnv-gnu for all compilation
    - still need `-lOpenCL` at linktime
- Submission:
  - submit job as usual (SLURM, aprun):
    - Use 1 MPI rank per node

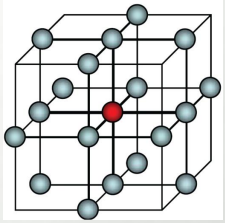
In conclusion...

- Hybrid multicore has arrived and is here to stay
  - Fat nodes are getting fatter
  - GPUs have leapt into the top500 and accelerated nodes

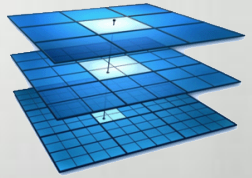


- Programming accelerators efficiently is hard
  - When done well can give good performance (Ludwig)
- Accelerator directives offer a good alternative
  - Attractive (and familiar) programming model
  - Open standards for portability
  - Use original Fortran, C and C++ codes





- Presented a strategy for porting large codes
  - The performance penalty is small
  - The portability and productivity bonuses are huge



- Directives play nicely with (some) other programming models
  - so you don't need to throw away your prize CUDA kernels

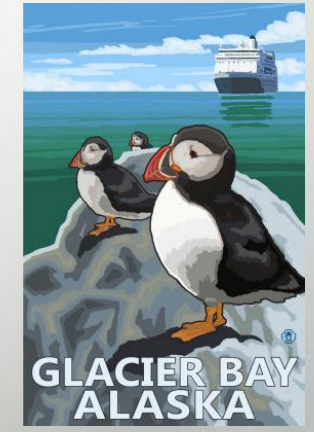
Tutorial overview

- Tutorial leads you through porting entire MultiGrid code
  - Structure resembles SciEng application in only 1500 lines
    - More useful than over-simplistic "Hello World" examples
  - Tutorial covers:
    - code preparation
    - profiling and scoping
    - steps to progressively port to GPU using OpenACC
  - Code examples and Makefiles provided
    - Both Fortran and C versions (no C for 01, 02)
    - `VERSION=00` Original CPU code
    - `VERSION=01` CPU version for profiling with CrayPAT
    - `VERSION=02` CPU version for variable scoping with Cray Reveal
    - `VERSION=03` First OpenACC kernel
    - `VERSION=04` OpenACC for all significant kernels
    - `VERSION=05` OpenACC for insignificant kernels as well
    - `VERSION=06` Data region to eliminate major data movements
    - `VERSION=07` Tuned OpenACC region
    - `VERSION=08` OpenACC interoperating with CUDA kernel

Acknowledgments

Thank you to those that helped us get to grips with directives:

- Cray Exascale Research Initiative Europe team
  - Harvey Richardson, Jason Beech-Brandt
  - Roberto Ansaloni
- EPCC Exascale Technology Centre team
  - Alan Gray
- Cray PE R&D team
  - Luiz DeRose, Heidi Poxon, Suzanne LaCroix, James Beyer, David Oehmke...
- ORNL team
  - John Levesque, Jeff Larkin
- OpenMP subcommittee



For further info, ahart@cray.com