OpenACC: extra topics and roadmap

Alistair Hart
Cray Exascale Research Initiative Europe

Contents of Talk

- OpenACC interoperability
- Debugging OpenACC applications
- Using Scientific Libraries with OpenACC
- OpenACC Roadmap

Interoperability

- **OpenACC is a complete programming model**
  - But there are still situations where it is useful to interface OpenACC code with other GPU programming models
- **Why might this be useful?**
  - You want to call accelerated scientific libraries from your code
    - without having to transfer data back and forth between the host
  - You want to call CUDA kernels from your code
    - also without unnecessary data transfers
  - You want to exploit Nvidia GPUdirect (or similar) to streamline communication of data between accelerators.
- **Interfacing requires access to the lower-level information**
  - Typically the GPU memory locations of OpenACC-created data arrays
  - The compiler normally hides this information from the user.

host_data directive

- **OpenACC runtime manages GPU memory implicitly**
  - user does not need to worry about memory allocation/free-ing
- **Sometimes it can be useful to know where data is held in device memory, e.g.:**
  - so a hand-optimised CUDA kernel can be used to process data already held on the device
  - so a third-party GPU library can be used to process data already held on the device (Cray libsci_acc, cuBLAS, cuFFT etc.)
  - so optimised communication libraries can be used to streamline data transfer from one GPU to another
- **host_data directive provides mechanism for this**
  - nested inside OpenACC data region
  - subprogram calls within host_data region then pass pointer in device memory rather than in host memory

Interoperability with CUDA

- **Why would you want to do this?**
- **Two situations:**
  - You have already ported an application to OpenACC
    - A few key kernels get improved performance using hand-tuned CUDA
      - (performance at the cost of reduced portability)
    - These CUDA kernels should process data that was already placed in GPU memory using OpenACC
  - Or, you have ported a few key kernels to the GPU using CUDA
    - but data movement costs outweigh the performance gain
    - OpenACC provides an efficient way of porting the remainder of the application

CUDA Interoperability

```
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

Streamlined Communications

- **To transfer data between GPUs typically means:**
  - "acc update host" to move data to local CPU memory
    - Plus implicit or explicit "acc wait" to ensure completion
  - transfer to remote CPU memory e.g. with MPI
    - Plus communications barrier to ensure completion
  - "acc update device" to move received data to GPU memory
    - Plus associated "acc wait"
- **A lot of synchronisation points**
  - User code has to manage these
  - Limits the scope for overlapping communications with computation
- **Some communications libraries can bypass this**
  - e.g. using Nvidia GPUDirect
  - Require a pointer to GPU memory to be passed to library
  - host_data provides a mechanism for doing this

Interoperability with Libraries

- **Why would you want to do this?**
  - You should always use libraries if they are available
    - A lot of effort goes into optimizing them
    - They are likely to use a lot more tricks that you have time/inclination to try
- **Examples of libraries:**
  - Cray libsci_acc
  - cuBLAS
  - cuFFT
  - ...
- **To use these with OpenACC code**
  - Place calls to the library inside host_data regions

What is Cray Libsci_acc?

- **Provide basic scientific libraries optimized for hybrid CPU and accelerator systems**
- **Independent to, but fully compatible with OpenACC**
- **Multiple use case support**
  - Get the base use of accelerators with no code change
  - Get extreme performance of GPU with or without code change
  - Extra tools for support of complex code
- **Incorporate the existing GPU libraries into Cray libsci**
- **Provide additional performance and usability**

Libsci_acc Example

- Starting with a code that relies on dgemm.
- The library will check the parameters at runtime.
- If the size of the matrix multiply is large enough, the library will run it on the GPU, handling all data movement behind the scenes.
- NOTE: Input and Output data are in CPU memory.

```
call dgemm('n','n',m,n,k,alpha,&
           a,lda,b,ldb,beta,c,ldc)
```

Libsci_acc Example

- If the rest of the code uses OpenACC, it’s possible to use the library with directives
- All data management performed by OpenACC
- Calls the device version of dgemm
- All data is in CPU memory before and after data region

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm_acc('n','n',m,n,k,&
               alpha,a,lda,&
               b,ldb,beta,c,ldc)

!$acc end host_data

!$acc end data
```

Libsci_acc Example

- Libsci_acc is a bit smarter that this
- Since 'a,' 'b', and 'c' are device arrays, the library knows it should run on the device
- So just dgemm is sufficient

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm    ('n','n',m,n,k,&
               alpha,a,lda,&
               b,ldb,beta,c,ldc)

!$acc end host_data

!$acc end data
```

The OpenACC runtime API

Alistair Hart
Cray Exascale Research Initiative Europe

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 also offers an extended runtime API**
  - set of library calls, names starting with cray_acc_
    - will need preprocessing away if not using OpenACC *and* 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

Runtime API for device selection and control

- **About the OpenACC-supporting accelerators**
  - What type of device will I use next? acc_get_device_type()
    - default from environment variable ACC_DEVICE_TYPE
  - What type of device should I use next? acc_set_device_type()

  - How many accelerators of specified type? acc_get_num_devices()
  - Which device of specified type will I use next? acc_get_device_num()
    - default from environment variable ACC_DEVICE_NUM
  - Which device of specified type should I use next? acc_set_device_num()

  - Am I executing on device of specified type? acc_on_device()

- **Initialising/shutting down accelerators:**
  - Initialise (e.g. to isolate time taken): acc_init()
  - Shut down (e.g. before switching devices): acc_shutdown()

OpenACC runtime API

- **Device selection and control API calls**
- **Advice:**
  - Don't use these runtime calls unless you really need to
    - The defaults are all sensible
    - All you need on a host with one accelerator (e.g. Cray XK family)
  - Maybe acc_init() to isolate device initialisation from performance timing
    - not needed for CCE anyway: automatically initialises at program launch

Runtime API for advanced memory control

- **These are for very advanced users**
- **Offer method to allocate and free device memory**
- **C/C++ only:**
  - void* acc_malloc (size_t);
  - void acc_free (void*);

- **Advice:**
- **If you just need to know the address of the memory used by OpenACC (to pass, for instance, to CUDA)**
  - then you don't need these
  - just use host_data directive instead (we'll talk about this later)

Runtime API for asynchronicity

- **Runtime API can be used to control asynchronicity**
  - Advice: this is probably the part of the API you are most likely to use
- **Waiting for stream of operations to complete**
  - acc_async_wait(handle)
  - duplicates functionality of !$acc wait(handle) directive
- **Waiting for all operations to complete**
  - acc_async_wait_all()
  - duplicates functionality of !$acc wait directive
- **Can also test for completion without waiting**
  - a single stream of operations: acc_async_test(handle)
  - all operations: acc_async_test_all()
  - no directive equivalent for these

Cray extended runtime API

- **These go beyond the current OpenACC standard**
  - only currently supported by CCE
  - using these can make the code non-functioning for pure CPU
    - so you will almost certainly need to pre-process the code

- **Allows some advanced control of device memory**

- **see man intro_openacc (with PrgEnv-cray loaded) for details**
  - and man openacc.examples

OpenACC v2.0

The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

CAPS

CRAY THE SUPERCOMPUTER COMPANY

NVIDIA.

PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

OpenACC v2.0

- **Progress report**
  - Technical report issued last November
  - OpenACC committee now finalising v2.0 of the standard
  - Expected to be formally launched shortly
- **Here we summarise the proposed additions:**
  - default(none) for parallel or kernels directives
    - so the programmer must be explicit about all data movements
  - unstructured data lifetimes
  - call support
  - async clause for wait directive
  - nested parallelism
  - data API routines
  - tile clause for loop directive

Unstructured data lifetimes

- **To keep data on the accelerator between two routines**
  - currently need a data region in a parent routine common to both
  - the arrays must be scope for that parent routine
- **This can be inconvenient...**
  - e.g. Fortran module is not currently USEd in the parent routine
- **... or impossible**
  - data is created/destroyed by separate constructor/destructor methods
- **Two new directives proposed:**
  - enter data
    - clauses: [present_or_]copyin, [present_or_]create, if
    - data then persists on the accelerator until code reaches a matching:
  - exit data
    - clauses: [present_or_]copyout, [present_or_]delete, if

Call support

- **v1.0 does not support calls within accelerator regions**
  - either compiler has to inline
    - makes it hard to understand and control the scheduling
  - or the user has to do it
    - destroying their application's calltree structure
  - it also limits the use of third-party libraries

- **New directive proposed:**
  - routine
  - clauses include:
    - bind: useful for cross-compiler linking
    - type: describes what partitioning has been used so far and what is available
    - nohost: the routine will never be called outside an accelerator region

async clause for wait directive

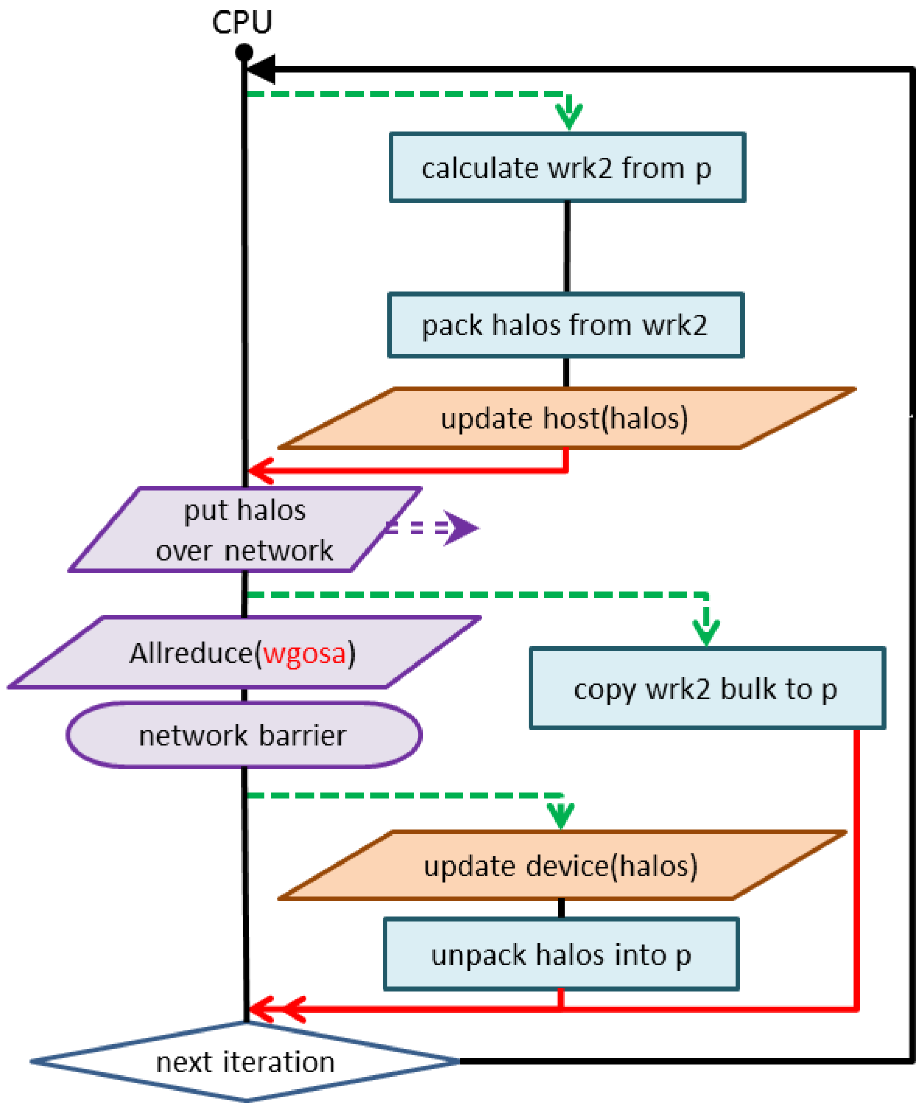
- **"When you have finished updating this array, separately pack halo buffers in the x and y directions"**
  - two kernels should start in separate async streams
    - but only after one async kernel finishes
  - Only way to do this in v1.0 is for host to wait on the first kernel
    - introduces extra (unwanted) synchronisation point

- **Proposed change:**
  - async clause for wait directive
    - async stream won't progress until
    - wait stream has completed
  - better task dependency tree
  - more scope for overlap
  - example:
    - after wait directive, know:
      - bulk updated on GPU
      - then halo buffers packed on GPU
      - then buffers transferred to CPU

```
!$acc parallel loop async(Sbulk)
<update bulk>

!$acc wait(Sbulk) async(Sxhalo)
!$acc parallel loop async(Sxhalo)
<pack xhalo>
!$acc update host(xhalo) async(Sxhalo)

!$acc wait(Sbulk) async(Syhalo)
!$acc parallel loop async(Syhalo)
<pack yhalo>
!$acc update host(yhalo) async(Syhalo)

<independent host code>
!$acc wait
```

async clause for wait directive

- **Parallel Himeno code**
  - removes need for sync after first stencil kernel
  - host has greater freedom to do other things
    - unfortunately, code too simple
    - no other tasks to overlap with



Nested parallelism

- **Currently can't have a parallel region inside another one**
  - or kernels
  - nested parallelism offers potential speed-up
    - if the hardware supports it (e.g. "dynamic parallelism" on Kepler)

- **Proposed change**
  - allow nested parallel and kernels regions
    - limited range of clauses for the nested regions:
      - present, private, firstprivate
      - because outermost one handles all the data movement etc.
  - if the hardware doesn't support nested parallelism
    - inner regions executed sequentially

Data API routines

- A set of API calls that are equivalent to directives
  - update
    - acc_update_device(), acc_update_host()
  - enter data
    - acc_[p]copyin, acc_[p]create
  - exit data
    - acc_[p]copyout, acc_[p]delete

- for those that prefer API calls

tile clause

- **v1.0 offers no way to tile loopnests**
  - to take advantage of multidimensional (2D/3D) partitioning
  - and/or use the cache clause
  - users have to do it by hand, or use vendor-specific directives

- **Proposed tile(n) clause**
  - applied to loop directive
  - tiles the following n (tightly-nested) loops
    - optionally, can specify tile sizes: tile(n:size1,size2,...,sizen)

  - scheduling optionally controlled using gang, worker, vector clauses on associated loop directive
    - default scheduling options are implementation dependent

OpenACC Roadmap – Areas for Improvement

- **Deep copy**
  - Similar to MPI system
  - Self describing structures
  - Multiple “images of structure”
- **Separate compilation units**
  - True calls, not call-site flattening
  - Orphaned loops
- **Dynamic Parallelism**
  - Not OpenMP style nested parallelism
- **C / C++ Multidimensional arrays**
  - VLAs
    - float f_array[10][10]
  - Ragged arrays
    - float **f_array
- **Better async system**
- **Noncontiguous memory**
- **Directive versions of extended runtime routines**
- **Multiple devices**

In summary

- **OpenACC provides a method for porting existing codes**
  - Fortran, C and C++
  - Minimal changes to the source code
  - In many cases, performance is good
- **What to take away:**
  - acc parallel loop
    - accelerates the loopnests
  - acc data
    - reduces data movements
  - vector_length(...), collapse
    - first things to try for performance optimisation
    - but only once you have eliminated data transfers

Finally...

- **Now you know as much as anyone about OpenACC**

- **Remember the Three Acc-s:**

  - **acc**urate:
    - make sure you have correctness measures (checksums)
    - and check them at every stage of the port

  - **acc**essible:
    - keeping the data in the right place is the biggest performance boost

  - **acc**elerate:
    - kernel optimisations come after this

- **Please ask for help if you need it**
  - CSCS has some of the most expert users of OpenACC in the world
  - Cray can help you as well