Recent Advances in Parallel Programming Languages: OpenACC

LRZ München

Mandes Schönherr,
CRAY CoE @ HLRS, Suttgart

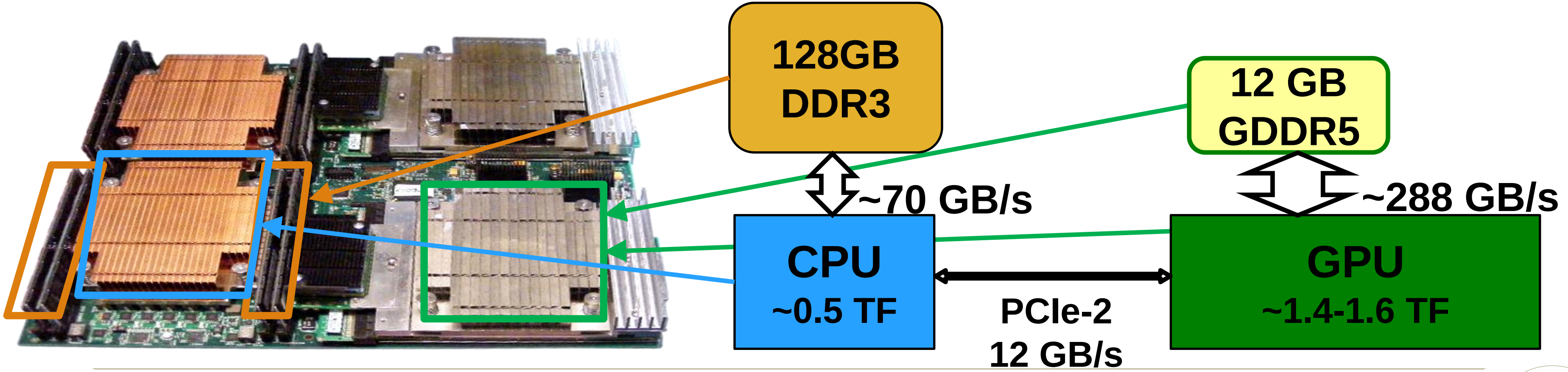
Agenda

- **A quick GPU refresher**
  - Hardware and programing models
- **OpenACC compared with OpenMP**
  - pragmas and OpenMP comparison
- **OpenACC 2.x/3.0**

A quick GPU refresher

How fast are current GPUs?

- **What should you expect?**
  - On a typical hybrid system (e.g. Cray XC):
    - Flop/s: GPU ~3x faster than a single CPU (using all 12 cores)
    - Memory bandwidth: GPU ~3.5x faster than CPU
  - These ratios are going to be similar in other systems
- **But, it is harder to reach peak performance on a GPU**
  - Your code needs to fit the architecture
  - You also need to factor in data transfers between CPU and GPU

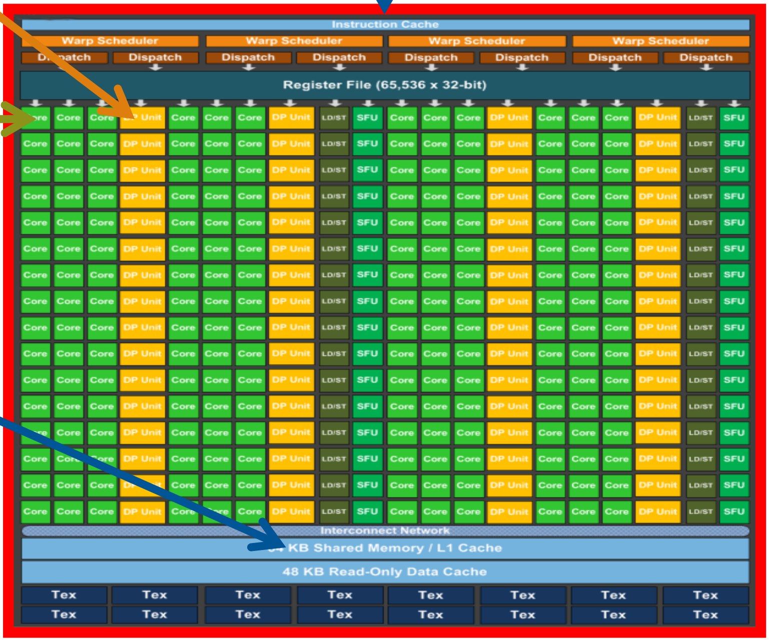


Nvidia K40 Kepler architecture (1)

- **Global architecture**
  - a lot of lightweight compute cores
    - 2880 SP plus 960 DP (ratio 3:1)
  - divided into 15 Streaming Multiprocessors (SMX)
  - SMXs operate independently of each other

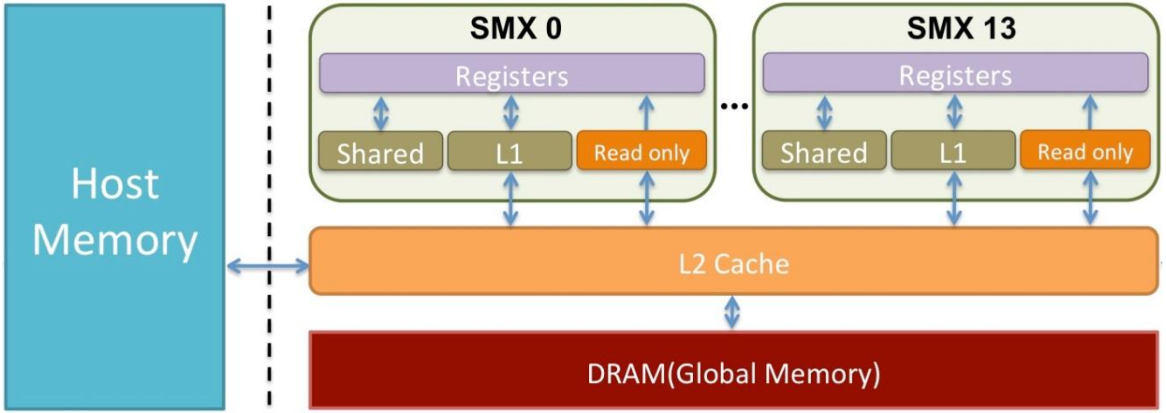
Nvidia K40 Kepler architecture (2)

- **SMX architecture**
  - many cores (192 SP plus 64 DP)
  - shared instruction stream
    - lockstep, SIMT execution of same ops
    - SMX acts like vector processor
    - warps of 32 entries
- **Memory hierarchy**
  - each core has private registers
    - fixed register file size
  - cores in an SMX share a fast L1 cache
    - 64KB, split between:
      - L1 cache and user-managed
  - large global memory
    - shared by all SMXs (cores)
    - 12GB; also some specialist memory



Memory model

- **Current GPUs have a weak memory model**
  - host and device have separate memories
    - different memory addresses, different data
  - there is no automatic synchronisation of the memory spaces
    - all **synchronisation** must be done **explicitly** by the host
    - directed either by the user or by the runtime (the compiler may help)



Program execution with a GPU

- **main program: host (CPU)**
  - Code on host either serially or in parallel with threads (e.g. OpenMP)
    - calculations that you want to be done on the CPU, e.g.
      - it is hard to parallelise for the GPU
      - there is not enough work to justify using the GPU
    - communication calls, e.g. MPI
    - control statements for the GPU, e.g.
      - memory management and data transfer on host and GPU
      - launch “kernels” on GPU
      - synchronisation

- **kernels (tasks): device (GPU)**
  - launched from the host
  - specially written for the GPUs, e.g. with
    - CUDA, OpenCL, Stream, hiCUDA, …
      - User need to rewrite kernels in quite low-level special language
      - Hard to write and debug
      - Hard to optimise for specific GPU
      - Hard to port to new accelerator
    - OpenACC
      - directive-based,
      - Based on original source code (easier to maintain/port/extend)

Kernels

- **GPU kernels are executed by many threads in parallel**
  - all threads execute the same code
    - perform the **same operations**, but on **different data**
  - can take different paths in the code
    - actually, they all take the same paths but some threads spin

  - each thread has a unique ID
  - this can be used to
    - select which data elements to process
    - make control decisions

- **Each kernel thread will be executed by a core on the GPU**

- **Threads are grouped together**
  - threads are grouped into "blocks" (or "gangs")
    - typically hundreds of threads per block
  - the group of blocks is called a "grid"

Kernel execution: threadblocks

- **each threadblock will execute on a single SMX**
  - you can have more threads than there are cores in an SMX
  - you really want this to happen
    - so the GPU has enough computational work
- **different threadblocks will execute on different SMXs**
  - several threadblocks can be executing on the same SMX
  - you really want this to happen
    - threadblocks will be swapped in and out of execution to hide memory latency
  - you have no control over this
    - so you cannot predict which order threadblocks execute in
    - nor is there any way to impose a full barrier within a kernel
- **threads within a threadblock <u>can</u> interact**
  - they can communicate data via a fast shared memory
  - you can synchronise <u>within</u> a threadblock

Kernel execution: warp

- **Threadblocks are divided into sets of 32 threads (warp)**
  - SMX is really a vector processor of width 32
  - groups of 32 cores act in lockstep, rather than independently
  - shares a single instruction stream with single program counter

- **Multiple warps in threadblock are executed in turn**
  - i.e. if there are more than 32 threads in the threadblock
- **Memory loads/stores are also done on a per-warp basis**
  - Loading/storing 32 consecutive memory addresses at once

- **So, really, the compiler is implementing your code using vector instructions**
  - This is not explicit in the CUDA programming model, but is crucial to gaining good performance from a GPU
    - whichever programming model you are using (it's a hardware thing)

What does this mean for the programmer?

- 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 SMX to hide memory latency
  - Not just 2880 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 the 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

OpenACC model

- **OpenACC is a specification for high-level compiler directives, expressing parallelism for accelerators**
  - Directives are comments in the code
    - automatically ignored by non-accelerating compiler
- **OpenACC support in CCE and PGI**
  - on Cray machines
    - load module, e.g. module load craype-accel-nvidia35
    - Use compiler wrapper, ftn for Fortran, cc for C, and CC for C++
- **OpenACC initiated by CRAY, CAPS, PGI, NVIDIA**
  - 1.0: Nov. 2011
  - 2.0: Jun. 2013
  - 2.5 and 3.0 in near future

First example

Matrix-vector multiplication

```
#pragma acc data copyin(a[0:n*m])
{
  ...
  #pragma acc data copyin(v[0:n]) \
                   copyout(x[0:n])
  {
    ...
    matvecmul( x, a, v, m, n );
    ...
  }
  ...
}
```

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){
#pragma acc parallel loop gang \
  pcopyin(a[0:n*m],v[0:n]) pcopyout(x[0:m])
  for( int i = 0; i < m; ++i ){
    float xx = 0.0;

#pragma acc loop worker reduction(+:xx)
    for( int j = 0; j < n; ++j )
      xx += a[i*n+j]*v[j];
    x[i] = xx;
  }
}
```

OpenACC compared with OpenMP

pragma by pragma

OpenACC to OpenMP: Compute constructs

OpenACC		OpenMP
`!$acc kernel`	Compiler finds parallelism	Not supported
`!$acc parallel`	Offload work	`!$omp target teams`
`!$acc loop gang`	schedule threads within grid	`!$omp distribute`
`!$acc loop worker`	schedule threads within thread block	Not supported
`!$acc loop vector`	schedule threads within warp	`!$omp simd`

OpenACC to OpenMP: Data regions

OpenACC		OpenMP
`!$acc data` **create/pcreate** **copyin/pcopyin** **copy/pcopy** **copyout/pcopyout** **present**	Manage data transfer allocateting, deallocating, and copying from and to the device pcopy* is alias for present_or_copy*	`!$omp target data` map(alloc:) map(to:) map(tofrom:) map(from:) Possible 4.1
!$acc update self **!$acc update device**	data movement in data environment	!$omp target update from !$omp target update to
!$acc enter/exit data	unstructured data lifetime	!$omp enter/exit target data (4.1)
!$acc host_data	interoperability with CUDA/ libs	Possible in 4.1

OpenACC to OpenMP: Separate compilation

OpenACC		OpenMP
`!$acc declare create`	declare global, static data	`!$omp declare target`
`!$acc declare device_resident`	Create device copy, but no allocation on host	Not supported
`!$acc declare link`	Link (pointer) on device to data on host	Not supported
`!$acc routine`	for function calls	`!$omp declare target`

OpenACC to OpenMP: Other

OpenACC		OpenMP
API routines	OpenACC functionality provided by function calls	Most supported in 4.1
`!$acc atomic`	atomic operations	Use regular OpenMP atomics
`!$acc cache`	advice to put objects to closer memory	Not supported
`!$acc … async(handle)` `!$acc wait(handle)`	asyncronous process, waiting	- Tasks in 4.1 - `depend/nowait` on `target` in 4.1

OpenACC to OpenMP: model approach

- **OpenACC**
  - aims for portable performance
  - Focus on directives for accelerators
  - Descriptive approach to parallel programming

- **OpenMP**
  - aims for programmability
  - More general definition of pragmas
  - Prescriptive approach to parallel programming

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
      - e.g. of asynchronicity acc_async_test_all()
      - e.g. initialization/finalization
        acc_shutdown(), acc_init() … can prevent delay in initializing the GPU
    - Data allocation and movement

Should I just wait for OpenMP4 support?

NO!

The knowlegde you gain, the analysis and restructuring you do is portable.

OpenACC 2.x/3.0

Deep Copy
Or
Type serialization

API / Directive Equivalence

- acc_init()
- acc_shutdown()

- acc_set_device_num()

- !$acc enter data copyin() async

- !$acc update device() async

- !$acc init(nvidia)
- !$acc shutdown

- !$acc set device(nvidia,num:1)

- acc_copyin_async()

- acc_update_device_async

Flat object model

- **OpenACC supports a "flat" object model**
  - Primitive types
  - Composite types without allocatable/pointer members

```
struct {
  int x[2]; // size 2
} *A;        // size 2
#pragma acc data copy(A[0:2])
```

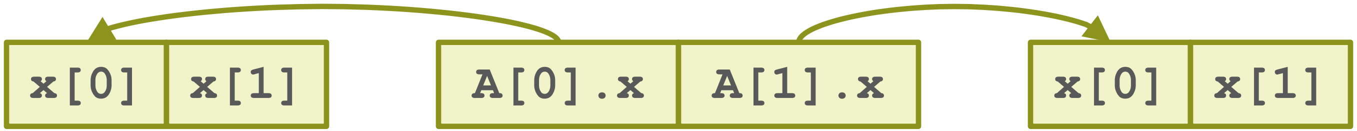
Host Memory:	A[0].x[0]	A[0].x[1]	A[1].x[0]	A[1].x[1]
Device Memory	dA[0].x[0]	dA[0].x[1]	dA[1].x[0]	dA[1].x[1]

Challenges with pointer indirection

- **Non-contiguous transfers**
- **More simply moving data hidden behind a pointer**
  - Fortran pointers have size information built in
  - C and C++ pointers …

```
struct {
   int *x; // size 2
} *A;     // size 2
#pragma acc data copy(A[0:2])
```

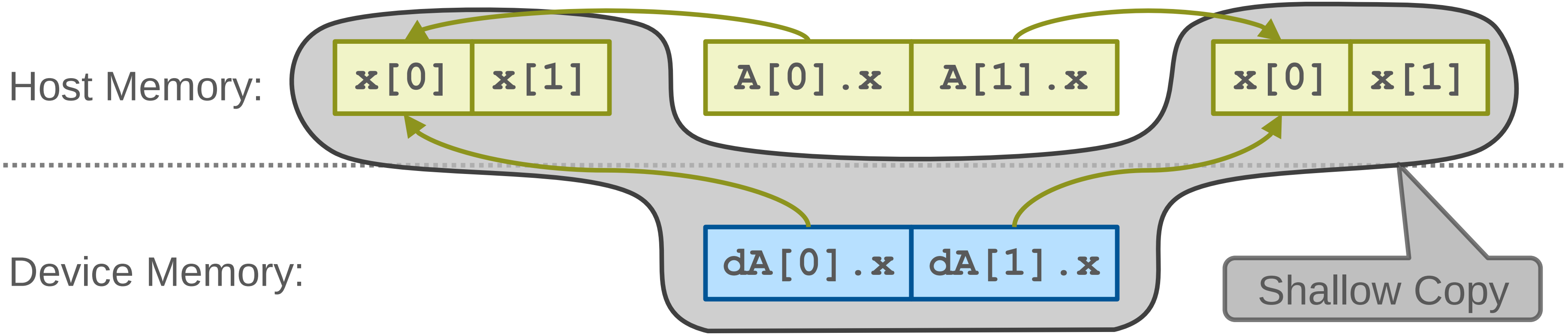
Host Memory:



Challenges with pointer indirection

- **Non-contiguous transfers**
- **More simply moving data hidden behind a pointer**
  - Fortran pointers have size information built in
  - C and C++ pointers …

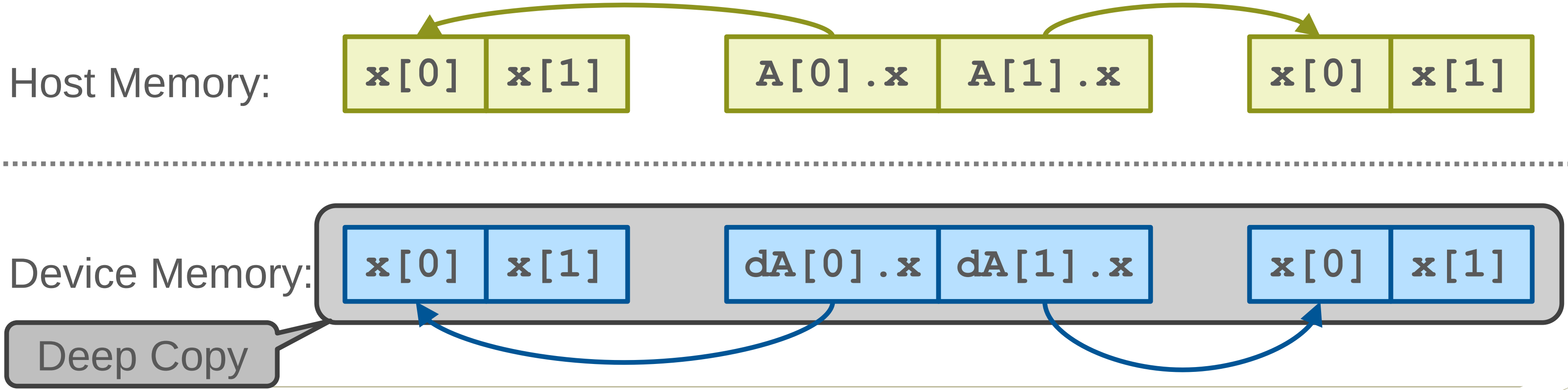
```
struct {
  int *x; // size 2
} *A;     // size 2
#pragma acc data copy(A[0:2])
```



What is deep copy?

- **Non-contiguous transfers**
- **More simply moving data hidden behind a pointer**
  - Fortran pointers have size information built in
  - C and C++ pointers …

```
struct {
  int *x; // size 2
} *A;     // size 2
#pragma acc data copy(A[0:2])
```



Todays possible deep-copy solutions

- **Re-write application**
  - Use “flat” objects
- **Manual deep copy**
  - Issue multiple transfers
  - Translate pointers
- **Compiler-assisted deep copy**
  - Automatic for Fortran
    - -hacc_model=deep_copy
    - Dope vectors are self describing
  - OpenACC extensions for C/C++
    - Pointers require explicit shapes

Appropriate for CUDA

Appropriate for OpenACC

Manual deep-copy

```
struct A_t {
  int n;
  int *x;       // size n
};
...
struct A_t *A; // size 2

/* shallow copyin A[0:2] to device_A[0:2] */
struct A_t *dA = acc_copyin( A, 2*sizeof(struct A_t) );
for (int i = 0 ; i < 2 ; i++) {
  /* shallow copyin A[i].x[0:A[i].n] to "orphaned" object */
  int *dx = acc_copyin( A[i].x, A[i].n*sizeof(int) );
  /* fix acc pointer device_A[i].x */
  cray_acc_memcpy_to_device( &dA[i].x, &dx, sizeof(int*) );
}
```

- **Currently works for C/C++**
  - Fortran programmers have to know the tricks
- **not usually practical**

Proposed new directives

shape

- Self-describing Structures
- Inform the compiler of the shape of the data behind the pointer

```
struct A {
        int n;
        float* x;
#pragma acc delclare shape(x[0:n])
};
```

policy

- Data Policies
- Develop policies for how the data should be relocated

```
struct A {
        int n;
        float* x;
#pragma acc declare shape(x[0:n])
#pragma acc policy(“boundary”) \
   update(x[0:1],x[n-1:1])
};
```

*Syntax and functionality subject to change

Proposed “shape” directives

```
struct A_t {
  int n;
  int *x;       // size n
#pragma acc declare shape(x[0:n])
};
...
struct A_t *A; // size 2
...
/* deep copy */
#pragma acc data copy(A[0:2])
```

```
type Foo
    real,allocatable :: x(:)
    real,pointer     :: y(:)
    !$acc declare shape(x)   ! deep copy x
    !$acc declare unshape(y) ! do not deep
copy y
end type Foo
```

- Each object must shape its own pointers
- Member pointers must be contiguous
- No polymorphic types (types must be known statically)
- Pointer association may not change on accelerator (including allocation/deallocation)
- Member pointers may not alias (no cyclic data structures)
- Assignment operators, copy constructors, constructors or destructors are not invoked

```
!$acc exit data
```

Sources of further information

- **Standards web pages:**
  - 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 swan (or raven) account
  - Fora: 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)