The use of OpenACC and OpenMP Accelerator directives with the Cray Compilation Environment (CCE)

Dr. James C. Beyer

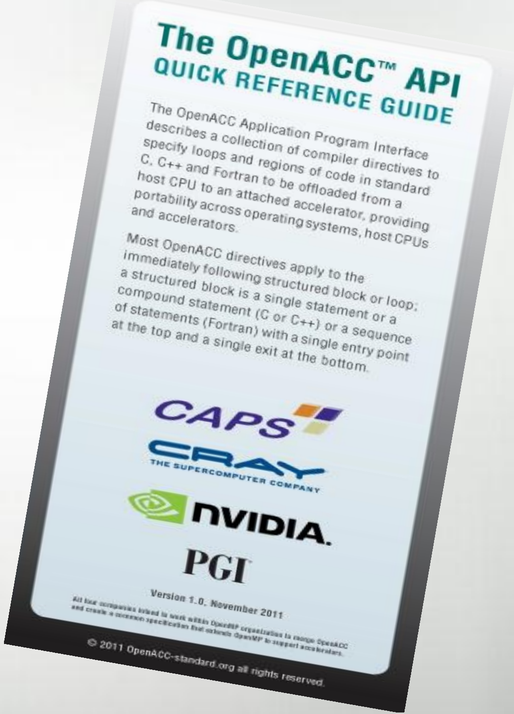
Contents

- Accelerator directives – what are they and why use them
- Difference between Accelerator Directives
- Cray Compilation Environment (CCE)
  - What does CCE do with *?
  - -hacc_model=
  - Extensions
    - Structure shaping
    - Deep copy
    - Selective deep copy
- Conclusions

Accelerator directives – what are they and why use them

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 www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 1.0 (November 2011)
- Next version: 2.0 (2013)
- Compiler support


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.

Version 1.0, November 2011

OpenMP

- A common directive programming model for (not so) shared memory systems
- Announced 15yrs ago
- Works with Fortran, C, C++
- Current version 3.1 (July 2011)
- Accelerator version (2013)
- Compiler support
    - http://openmp.org/wp/openmp-compilers/

OpenACC and OpenMP 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 and OpenMP Memory model

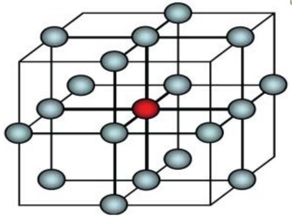
- Memory spaces on the host and device distinct
  - Different locations, different address space
  - Data movement performed by host using runtime library calls that explicitly move data between the separate
- GPUs have a weak memory model
  - No synchronisation between different execution units (SMs)
    - 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.

Why Directives?

- Most important hurdle for widespread adoption of accelerated computing ~~in HPC~~ is programming difficulty.
  - ~~Proprietary languages~~
  - Need portability across platforms
    - AMD, Intel, Nvidia, etc.
    - Device and host
  - Multi-language
  - Single code base
  - Multi-vendor support

Motivating example: Reduction

- Sum elements of an array
- Original Fortran code



```
a=0.0

do i = 1,n
 a = a + b(i)
end do
```

The reduction code in optimized CUDA

```
template<class T>
struct SharedMemory {
   __device__ inline operator      T*() {
      extern __shared__ int __smem[];
      return (T*)__smem;
   }
   __device__ inline operator const T*() const {
      extern __shared__ int __smem[];
      return (T*)__smem;
   }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void reduce6(T *g_idata, T *g_odata, unsigned int n) {
   T *sdata = SharedMemory<T>();

   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
   unsigned int gridSize = blockSize*2*gridDim.x;

   T mySum = 0;
   while (i < n)    {
      mySum += g_idata[i];
      if (nIsPow2 || i + blockSize < n)
         mySum += g_idata[i+blockSize];
      i += gridSize;
   }
   sdata[tid] = mySum;
   __syncthreads();

   if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum + sdata[tid + 256]; }
__syncthreads(); }
   if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum + sdata[tid + 128]; }
__syncthreads(); }
   if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum + sdata[tid +  64]; }
__syncthreads(); }
```

```
  if (tid < 32) {
     volatile T* smem = sdata;
     if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
     if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
     if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
     if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
     if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
     if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
  }
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b) {
  int *b_d;
  const int b_size = *n;

  cudaMalloc((void **) &b_d , sizeof(int)*b_size);
  cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);
  dim3 dimBlock(128, 1, 1);
  dim3 dimGrid(128, 1, 1);
  dim3 small_dimGrid(1, 1, 1);
  int smemSize = 128 * sizeof(int);
  int *buffer_d;
  int small_buffer[4],*small_buffer_d;

  cudaMalloc((void **) &buffer_d , sizeof(int)*128);
  cudaMalloc((void **) &small_buffer_d , sizeof(int));
  reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d, b_size);
  reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize>>>(buffer_d, small_buffer_d,128);
  cudaMemcpy(small_buffer, small_buffer_d, sizeof(int), cudaMemcpyDeviceToHost);

  *a = *small_buffer;

  cudaFree(buffer_d);
  cudaFree(small_buffer_d);
  cudaFree(b_d);
}
```

The reduction code in OpenACC™ API

- Compiler does the work:
  - Identifies parallel loops within the region
  - Determines the kernels needed
  - Splits the code into accelerator and host portions
  - Workshares loops running on accelerator
    - Make use of MIMD and SIMD style parallelism
  - Data movement
    - allocates/frees GPU memory at start/end of region
    - moves of data to/from GPU

```
!$acc data present(a,b)

a = 0.0

!$acc update device( a )
!$acc parallel
!$acc loop reduction(+:a)

do i = 1,n
  a = a + b(i)
end do

!$acc end parallel
!$acc end data
```

The reduction code in OpenMP™ API

```
a = 0.0

!$omp target update to( a )
!$omp target
!$omp team
!$acc distribute reduction(+:a)
do i = 1,n
  a = a + b(i)
end do
!$omp end distribute
!$omp end team
!$omp end target
```

```
a = 0.0

!$omp target update to( a )
!$omp target
!$omp parallel
!$acc do reduction(+:a)
do i = 1,n
  a = a + b(i)
end do
!$omp end do
!$omp end parallel
!$omp end target
```

Difference between Accelerator Directives

OpenACC compared to OpenMP

OpenACC 1

- Parallel (offload)
  - Parallel (multiple “threads”)
- Kernels
- Data
- Loop
- Host data
- Cache
- Update
- Wait
- Declare

OpenMP

- Target
- Team/Parallel
-
- Target Data
- Distribute/Do/for
-
-
- Update
-
- Declare

OpenACC compared to OpenMP continued

OpenACC 2	OpenMP
enter data	
exit data	
data api	
routine	declare target
async wait	
parallel in parallel	Parallel in parallel or team
tile	

OpenACC compared to OpenMP continued

OpenACC

-
-
-
-
-
-
-
-
-

OpenMP

- Atomic
- Critical sections
- Master
- Single
- Tasks
- barrier
- get_thread_num
- get_num_threads
- …

OpenMP async

- Target does NOT take an async clause!
  - Does this mean no async capabilities?
- OpenMP already has async capabilities -- Tasks
  - !$omp task
  - #pagma omp task
- Is this the best solution?

Cray Compilation Environment (CCE)

OpenACC in CCE

- man intro_openacc
- Which module to use, craype-accel-nvidia35
  - Kepler hardware
- Forces dynamic linking
- Single object file
- Whole program
- Messages/list file
- Compiles to PTX not cuda
- Debugger sees original program not cuda intermediate

-hacc_model=

- auto_async_(none | kernel | all)
- [no_]fast_addr
- [no_]deep_copy

OpenACC async clause

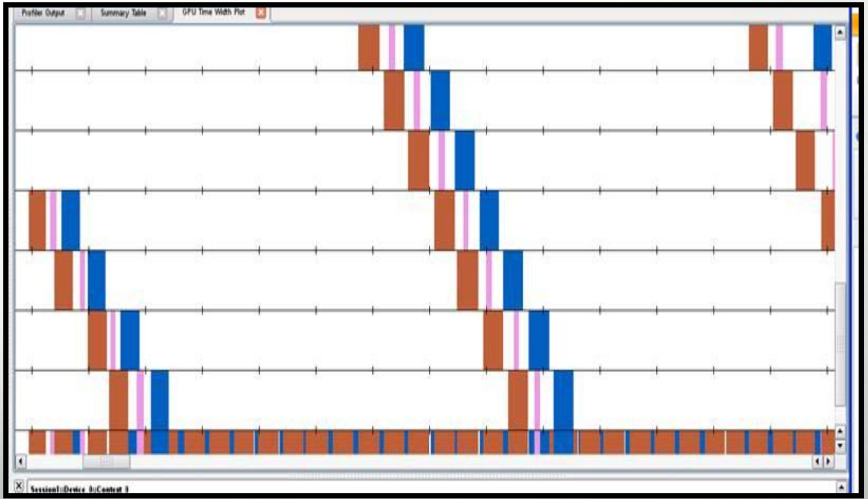
- async(handle): like CUDA streams
  - allows overlap of tasks on GPU
    - PCIe transfers in both directions
    - Plus multiple kernels (up to 16 with Fermi)
  - streams identified by handle
    - tasks with same handle execute sequentially
    - can wait on one, more or all tasks
      - OpenACC API also allows completeness check
- First attempt, a simple pipeline:
  - processes array, slice by slice
    - copy data to GPU, process, bring back to CPU
      - very complicated kernel operation here!
  - should be able to overlap 3 streams at once
    - use slice number as stream handle in this case
      - runtime MODs it back into allowable range
  - Can actually overlap more than three stream
    - No benefit on this test

```
INTEGER, PARAMETER :: Nvec = 10000, Nchunks = 10000

REAL(kind=dp) :: a(Nvec,Nchunks), b(Nvec,Nchunks)
!$acc data create(a,b)
DO j = 1,Nchunks
!$acc update device(a(:,j)) async(j)
!$acc parallel loop async(j)
  DO i = 1,Nvec
    b(i,j) = SQRT(EXP(a(i,j)*2d0))
    b(i,j) = LOG(b(i,j)**2d0)/2d0
  ENDDO
!$acc update host(b(:,j)) async(j)
ENDDO
!$acc wait
!$acc end data
```

OpenACC async results

- Execution times (on Cray XK6):
  - CPU: 3.98s
  - OpenACC, blocking: 3.6s
  - OpenACC, async: 0.82s
  - OpenACC, full async: 0.76s
- NVIDIA Visual profiler:
  - time flows to right, streams stacked vertically
    - red: data transfer to GPU
    - pink: computational kernel on GPU
    - blue: data transfer from GPU
  - vertical slice shows what is overlapping
    - only 7 of 16 streams fit in window
    - collapsed view at bottom
  - async handle modded by number of streams
    - so see multiple coloured bars per stream



OpenMP async example

- `target data map(alloc:a,b)`
  - allocates space for a and b
- `task depend(out:a(:,j))`
  `target update to(a(:,j))`
  - Copy host value of a(:,j) to device
  - Start async dependency chain on data
- `task depend(in:a(:,j)) depend(out:b(:,j))`
  `target team distribute`
  - execute loop across TB
  - Wait on update, start dependency chain on b
- `task depend(out:b(:,j))`
  `target update from(b(:,j))`
  - Copy device value of b(:,j)
  - Wait on compute kernel
- `taskwait`
  - Force all tasks to complete before data is removed from device

```
INTEGER, PARAMETER :: Nvec = 10000, Nchunks = 10000
REAL(kind=dp) :: a(Nvec,Nchunks), b(Nvec,Nchunks)

!$omp target data map(alloc:a,b)

DO j = 1,Nchunks

!$omp task depend(out:a(:,j))
!$omp target update to(a(:,j))
!$omp end task

!$omp task depend(in:a(:,j)) depend(out:b(:,j))
!$omp target team distribute
  DO i = 1,Nvec
    b(i,j) = SQRT(EXP(a(i,j)*2d0))
    b(i,j) = LOG(b(i,j)**2d0)/2d0
  ENDDO
!$omp end target team distribute
!$omp end task

!$omp task depend(out:b(:,j))
!$omp target update from(b(:,j))
!$omp end task

ENDDO

!$omp taskwait

!$omp end target data
```

Accel_mode example

Example code

```
!$acc data copy( a, b )
do i = 1, n
!$acc parallel loop
    do j = 1, m
        a(i) = func_j(j,a,b)
    end do
!$acc parallel loop
    do j = 1, m
        b(i) = func_j(j,b,a)
    end do
end do
```

What happens with –haccel_mode=

- auto_async_none
- auto_async_kernel
- auto_async_all

Accel_mode example 2

Example code

```
!$acc data copyout( a )
do i = 1, n
   a(i) = func_i(i)
!$acc update device( a(i) )
!$acc parallel loop
    do j = 1, m
        a(i) = func_j(j,a)
    end do
end do
```

What happens with –haccel_mode=

- auto_async_none
- auto_async_kernel
- auto_async_all

Accel_mode example

Example code

```
!$acc data copyout( a )
do i = 1, n
   a = func_i(i)
!$acc update device( a )
!$acc parallel loop
    do j = 1, m
        a = func_j(j,a)
    end do
end do
```

What happens with –haccel_mode=

- fast_addr

What does CCE do with OpenACC constructs

- Parallel/kernels
  - Flatten all calls
  - Identify kernels (kernels construct)
  - Package code for kernel
  - Generate PTX code for packaged code
  - Insert data motion to and from device
  - Insert kernel launch code
  - Automatic vectorization is enabled (!$acc loop vector)
- Update
  - Implicit !$acc data present(*obj*)
  - For known contiguous memory
    - Transfer (Essentially a CUDA memcpy)
  - Not contiguous memory
    - Pack into contiguous buffer
    - Transfer contiguous
    - Unpack from contiguous buffer
- Loop
  - Gang
    - Thread Block (TB)
  - Worker
    - warp
  - Vector
    - Threads within a warp or TB
  - Automatic vectorization is enabled
  - Collapse
    - Will only rediscover indices when required
  - Independent
    - Turns off safety/correctness checking for work-sharing of loop
  - Reduction
    - Nontrivial to implement
    - Does not use multiple kernels

What does CCE do with OpenACC constructs part 2

- Cache
  - Create shared memory “copies” of objects
    - Objects are sized according to directive reuse size
    - Loop Cache (a[i-1:3]) shared_a[3*vector_wide]
    - Generate a shared copy of array that is sized by the users directive and the subsequent strip mined loop.
  - Generate copy into shared memory objects
  - Generate copy out of shared memory objects

Partitioning clause mappings

1. !$acc loop gang : across thread blocks
2. !$acc loop worker : across warps within a thread block
3. !$acc loop vector : across threads within a warp

1. !$acc loop gang : across thread blocks
2. !$acc loop worker vector : across threads within a thread block

1. !$acc loop gang : across thread blocks
2. !$acc loop vector : across threads within a thread block

1. !$acc loop gang worker: across thread blocks and the warps within a thread block
2. !$acc loop vector : across threads within a warp

1. !$acc loop gang vector : across thread blocks and threads within a thread block

1. !$acc loop gang worker vector : across thread blocks and threads within a thread block

Partitioning clause mappings (cont)

You can also force things to be within a single thread block:

1. !$acc loop worker : across warps within a single thread block
2. !$acc loop vector : across threads within a warp

1. !$acc worker vector : across threads within a single thread block

1. !$acc vector : across threads within a single thread block

Extended OpenACC runtime routines

Version 1. 0

```
/* takes a host pointer */
void* cray_acc_create( void* , size_t );
void  cray_acc_delete( void* );
void* cray_acc_copyin( void*, size_t );
void  cray_acc_copyout( void*, size_t );
void  cray_acc_updatein( void*, size_t );
void  cray_acc_updateout( void*, size_t );
int   cray_acc_is_present( void* );
int   cray_acc_is_present_2( void*, size_t);
void *cray_acc_deviceptr( void* );

/* takes a device and host pointer */
void  cray_acc_memcpy_device_host( void*, void*, size_t );
/* takes a host and device pointer */
void  cray_acc_memcpy_host_device( void*, void*, size_t );

/* Takes a pointer to an implementation defined type */
bool cray_acc_get_async_info( void *, int )
```

Version 2. 0

Porting code to OpenACC

1) Identify parallel opportunities
2) For each parallel opportunity
   1) Add OpenACC Parallel Loop(s)
   2) Verify correctness
   3) Avoid data clause when possible, use present_or_* when required
3) Optimize “kernel” performance (how?)
   1) Add additional Acc Loop directives
   2) Add tuning clause/directives (Collapse, Cache, Num_gangs, num_workers, vector_length, …)
   3) Algorithmic enhancements/code rewrites*
4) Try fast address option

When making changes verify correctness often!

You cannot verify correctness too often!

Porting code to OpenACC

5) Add data regions/updates
   1) Try to put data regions as high in the call chain as profitable
   2) Working with one variable at a time can make things more manageable
   3) To identify data correctness issues can add excessive updates and remove them verifying correctness.
6) Try auto async all
   1) Auto async kernel is default
7) Add async clauses and waits
   1) If synchronization issues are suspected, try adding extra waits and slowly remove them.

When making changes verify correctness often!

You cannot verify correctness too often!

OpenACC correctness hints

- All parallel regions should contain a loop directive
- Fortran assumed size (A(*)) and C pointers must be shaped
- Always use ':' when shaping with an entire dimension (i.e. A(:,1:2))
- Host_data probably requires waits when combined with auto_async_(kernels|all)
  - Should start with auto_async_none
- Update (*) if(is_present(*)) can make code more composable

Tips for OpenMP

- Pretty much the analog of OpenACC tips!
- Start with “target team distribute”
- …

Extensions

- Deep copy
- Selective deep copy
- Structure shaping

Flat object model

- OpenACC supports a “flat” object model
  - Primitive types
  - Composite types without allocatable/pointer members

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

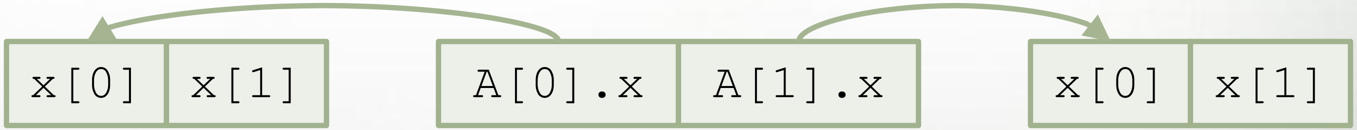
Host Memory:	A[0].x[0]	A[0].x[1]	A[1].x[0]	A[1].x[1]
Device Memory:	dA[0].x[0]	dA[0].x[1]	dA[1].x[0]	dA[1].x[1]

Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

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

Host Memory:

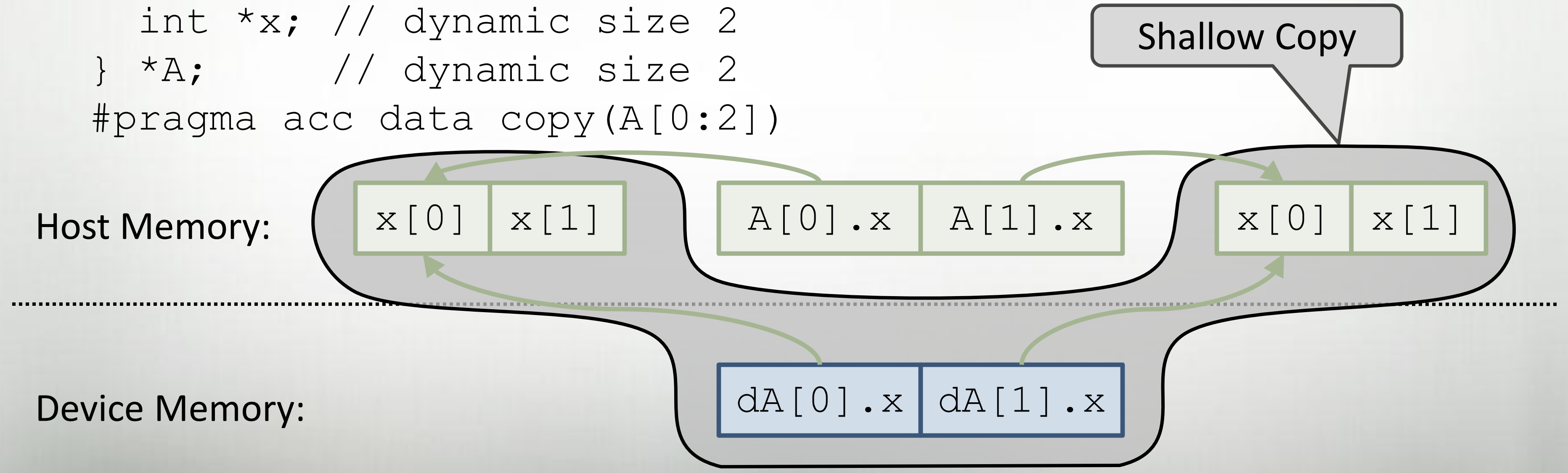


Device Memory:

Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

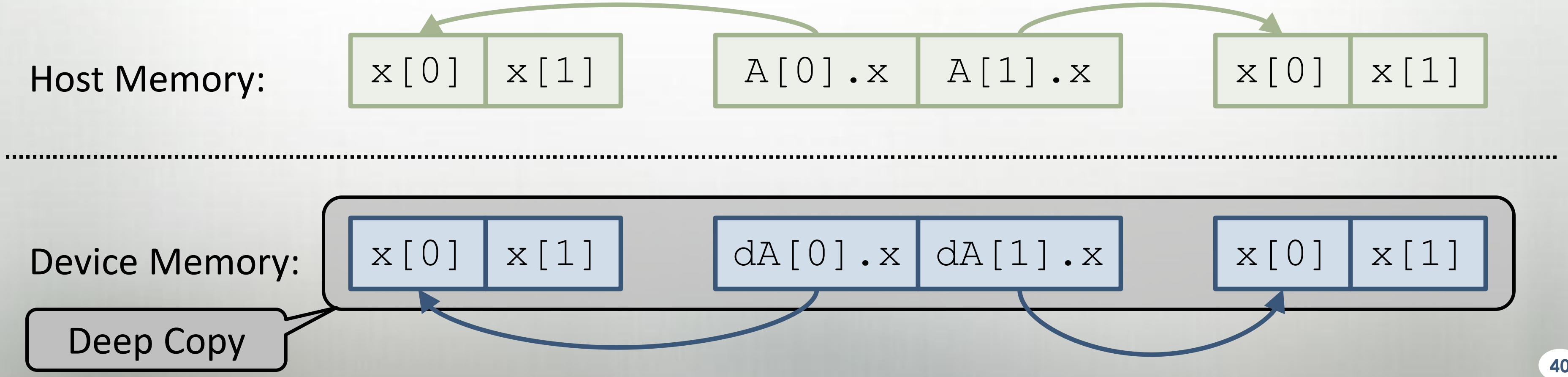
```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy(A[0:2])
```



Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

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



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_models=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;       // dynamic size n
};
...
struct A_t *A; // dynamic 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 */
  acc_memcpy_to_device( &dA[i].x, &dx, sizeof(int*) );
}
```

- Currently works for C/C++
- Portable in OpenACC 2.0, but not usually practical

Automatic Fortran deep-copy

```
type A_t
   integer,allocatable :: x(:)
end type A_t
...
type(A_t),allocatable :: A(:)
...
! shallow copy with -hacc_model=no_deep_copy (default)
!    deep copy with -hacc_model=deep_copy
!$acc data copy(A(:))
```

- No aliases on the accelerator
- Must be contiguous
- On or off – no “selective” deep copy
- Only works for Fortran

Semi-automatic C/C++ deep-copy

```
typedef struct {
  int *iptr;
} iptr_t;
iptr_t a;
a.iptr = malloc(8);
acc_copyin( a.iptr, 8 );
...
! shallow copy with -hacc_model=no_deep_copy (default)
!    deep copy "fixup" with -hacc_model=deep_copy
#pragma acc data copy( a )
```

- a.iptr is found on device so fixup value with device pointer
- If object is not present than no fixup and no error, “user selective”

Proposed “member shape” directives

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

- Each object must shape it’s 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

Member-shape directive examples

```
extern int size_z();
int size_y;
struct Foo
{
   double* x;
   double* y;
   double* z;
   int     size_x;
   // deep copy x, y, and z
   #pragma acc declare shape(x[0:size_x], y[1:size_y-1], z[0:size_z()])
};
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
```

Member Shape Status

- Library
  - Support for type descriptors
- Compiler
  - Automatic generation of type descriptors for Fortran
    - Compiler flag to enable/disable deep copy
    - Released in CCE 8.1
    - Significant internal testing, moderate customer testing
  - Directive-based generation of type descriptors for C/C++
    - Planned for release in CCE 8.2
    - Limited preliminary internal testing
- Language
  - Committee recognizes the utility and need
  - Will revisit after OpenACC 2.0

Conclusions

- Directive based programming models are progressing
- OpenACC