Tricks, Tips, and Timings: The Data Movement Strategies You Need to Know

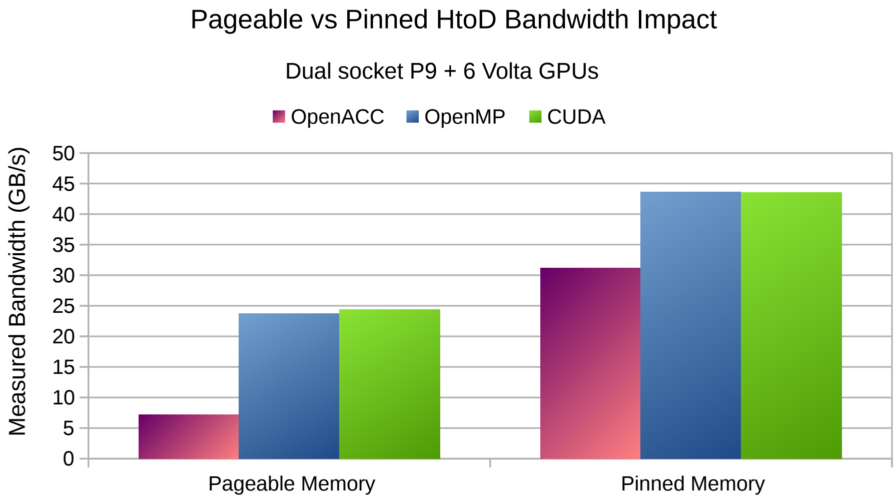
David Appelhans

GPU Technology Conference
March 26, 2018

Introduction

- My role: readying applications for SUMMIT and SIERRA supercomputers(past 3 years).
- Talk is a summary of data movement techniques, especially when working with NVLINK:
  - Importance of pinned memory. (Interoperability, CUDA+OpenMP+OpenACC)
  - Zero-copy tricks. (Interoperability, CUDA+OpenMP)
  - Dealing with nested data structures. (Efficiency, CUDA)
- All code examples are available on my public Github page. https://github.com/dappelha/gpu-tips/nvtx

Motivation: Why you should pin your memory



Hint: make sure your task starts in the appropriate socket: taskset -c 0 ./test

Pinned Memory Option 1:

Use CUDA Fortran[1] pinned attribute to pin at allocation time,

```
real(kind=8), pinned, allocatable :: p_A(:)
allocate( p_A(N) )
!$omp target data map(alloc:p_A)
do i=1,samples
  !$omp target update to(p_A)
    ...
enddo
```

Can also check success of pinning:

```
logical :: pstat
allocate( p_A(N), pinned=pstat)
if (.not. pstat) print *, "ERROR: p_A was not pinned"
```

[1] PGI and XLF compilers both support CUDA Fortran, so the pinned attribute can easily be combined with directives.

PINNED MEMORY OPTION 2:

Pin already allocated memory,[2]

```
use, intrinsic :: iso_c_binding
use cudafor
real, pointer, contiguous :: phi(:,:)
allocate( phi(dim1, dim2) ) ! phi can also be pointer passed from C++
istat = cudaHostRegister(C_LOC(phi(1,1)), sizeof(phi), cudaHostRegisterMapped)

!$acc enter data create(phi)
do i=1,samples
   !$acc update self(phi)
   ...
enddo
```

Warning: act of pinning memory is very slow. Memory should only be pinned if it is going to be used for data transfers.

[2]This technique is especially useful if the memory was allocated outside the developers control, for example in a C++ calling routine.

OpenACC Interoperabilty Warning 1

You must use the flag -ta=tesla:pinned in order for OpenACC to benefit from pinned memory.

1. *Compiling* with the flag -ta=tesla:pinned forces **all** memory to be pinned memory. This is a big hammer approach.
2. *Linking* the final executable with -ta=tesla:pinned causes the OpenACC runtime to check if an array is already pinned. This gives fine grain user control.

OpenACC Interoperabilty Warning 2

The OpenACC runtime uses a memory pool on the device to save from repeated allocation/deallocation of device memory. Can cause trouble when mixing CUDA with OpenACC.

```fortran
integer :: N = 8*gigabyte
real(kind=8), allocatable :: A(:)
real(kind=8), device, allocatable :: d_A(:)
allocate( A(N) )
!$acc enter data create(A)
!$acc exit data delete(A) ! <--not truly free'd unless PGI_ACC_MEM_MANAGE=0
allocate( d_A(N) )  ! <---- can then run out of device memory
```

To disable this optimization, set the environment flag PGI_ACC_MEM_MANAGE=0 and the runtime will free the data at the exit data.

Uses of Zero Copy

Zero copy refers to accessing host resident pinned memory directly from a GPU without having to copy the data to the device beforehand (i.e. there are zero device copies).

- Quick overlap of data movement and kernel compute (unified/managed memory is better for this purpose)
- Large arrays where only small percent of data is accessed in random pattern.
- All data is accessed, but read/write pattern is strided/not coalesced.
- Efficiently populating components of a structure, avoiding the overhead of many copy API calls by using GPU threads to fetch data directly.

CUDA Zero Copy Setup

To set up zero copy of a basic array in Fortran, use a CUDA API to get a device pointer that points to the pinned host array, and then associate a fortran array with that C device pointer, specifying the Fortran array attributes.

```
use iso_c_binding   ! provides c_f_pointer and C_LOC
! zero copy pointers for psib
type(C_DEVPTR)                          :: d_psib_p
real(adqt), device, allocatable :: pinned_psib (:,:,:)

! sets up zero copy of psib on device.
 istat = cudaHostGetDevicePointer(d_psib_p, C_LOC(psib(1,1,1)), 0)
! Translate that C pointer to the fortran array with given dimensions
call c_f_pointer(d_psib_p, pinned_psib, [QuadSet%Groups, Size%nbelem, QuadSet%NumAngles] )
```

OpenMP Zero Copy Example

Only requires CUDA pinned array and OpenMP is_device_ptr clause.

```
real (kind=8), pinned, allocatable :: A (:,:) ,At (:,:)
allocate ( A(nx,ny), At(ny,nx) )

! Transpose in the typical way:
!$omp target enter data map(alloc:A,At)
call transpose (A,At,nx,ny)
!$omp target update from(At)
!$omp target exit data map(delete:At)

! Ensure device has finished for accurate benchmarking
ierr = cudaDeviceSynchronize()

! Transpose using zero copy for At.
! At is no longer mapped--is_device_ptr(At) will
! allow addressing host pinned memory (zero copy)
call transpose_zero_copy(A,At,nx,ny)
```

continued on next slide

OpenMP Zero Copy Example Continued

```
subroutine transpose_zero_copy(A,At,nx,ny)
  ! example of strided writes to an array that lives on the host
  implicit none
  real(kind=8), intent(in) :: A(:,:)
  real(kind=8), intent(out) :: At(:,:)
  integer, intent(in) :: nx, ny
  integer :: i,j
  !$omp target teams distribute parallel do is_device_ptr(At)
  do j=1,ny
     do i=1,nx
        At(j,i) = A(i,j)
     enddo
  enddo
  return
end subroutine transpose_zero_copy
```

OpenMP Zero Copy Transpose

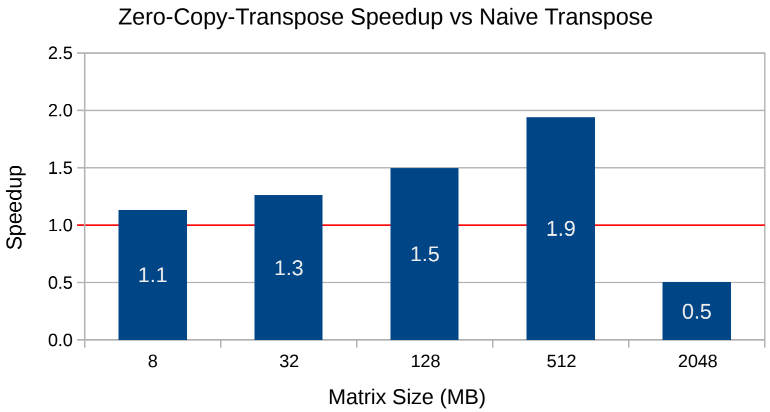


Figure : Power9 + V100 results of doing a traditional matrix transpose and then copying back from GPU vs doing the transpose directly into pinned host memory.

Nested Data Structures

Motivation:

```
subroutine my_kernel_to_port (...)
   ...
  element(id)%val(n) = element(id)%x(n)*element(id)%y(n)
   ...
end subroutine
```

Production codes often have dynamic structures with dynamic components.

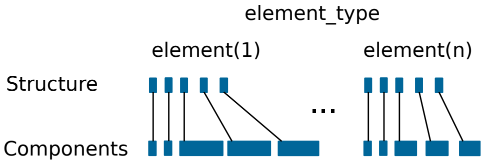
- Flattening data structures is messy (index arrays required for unstructured data) and invasive.
- Would like to keep nested references in compute kernel for portability.
- Often only parts of the data structure need to be used on the GPU.

Nested Data Structures

Two Topics:

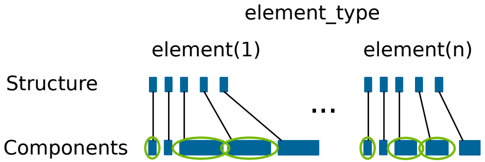
- **How do you make them referencable on the device?**
- How do you efficiently move data into them?

Often only parts of the data structure need to be used on the GPU



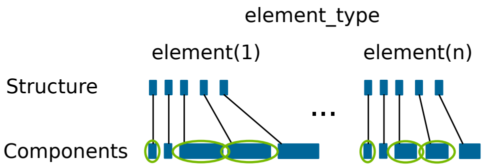

```fortran
type, public :: element_type
  integer                                   :: Nnodes
  real(kind=8)                              :: volume
  real(kind=8), allocatable, pinned :: x(:)
  real(kind=8), allocatable, pinned :: y(:)
  real(kind=8), allocatable, pinned :: val(:)
  real(kind=8), allocatable             :: old(:)
end type element_type
```

Often only parts of the data structure need to be used on the GPU




```
type, public :: element_type
  integer                                   :: Nnodes
  real (kind=8)                             :: volume
  real (kind=8), allocatable , pinned :: x (:)
  real (kind=8), allocatable , pinned :: y (:)
  real (kind=8), allocatable , pinned :: val (:)
  real (kind=8), allocatable            :: old (:)
end type element_type
```

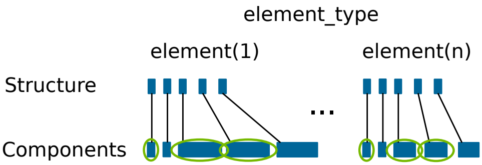
Often only parts of the data structure need to be used on the GPU




```
type, public :: element_type
   integer                                :: Nnodes
   real(kind=8)                           :: volume
   real(kind=8), allocatable , pinned :: x(:)
   real(kind=8), allocatable , pinned :: y(:)
   real(kind=8), allocatable , pinned :: val(:)
   real(kind=8), allocatable          :: old(:)
end type element_type
```

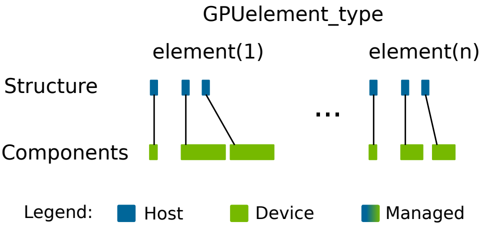
Can create a skinny version of the data structure with components that are device variables.

Often only parts of the data structure need to be used on the GPU




```fortran
type, public :: element_type
   integer                                    :: Nnodes
   real (kind=8)                              :: volume
   real (kind=8), allocatable , pinned :: x (:)
   real (kind=8), allocatable , pinned :: y (:)
   real (kind=8), allocatable , pinned :: val (:)
   real (kind=8), allocatable           :: old (:)
end type element_type
```

Can create a skinny version of the data structure with components that are device variables.




```fortran
type, public :: GPUelement_type
   integer                                    :: Nnodes
   real (kind=8), device, allocatable :: x (:)
   real (kind=8), device, allocatable :: y (:)
   real (kind=8), device, allocatable :: val (:)
end type GPUelement_type
```

This gives a way to loop through the structure and allocate device components while on the host:

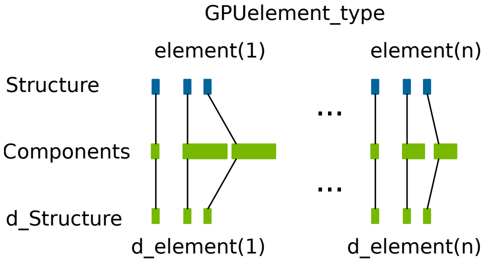
```
! can allocate on the host
do id=1,Nelements
   Nnodes = element(id)%Nnodes
    allocate (element(id)% x(Nnodes) )
enddo
```

but still cannot use the structure in a device kernel.

```
attributes(global) subroutine cuda_kernel (...)
    ...
   x = element(id)%x ! <-- invalid reference of element
    ...
end subroutine cuda_kernel
```

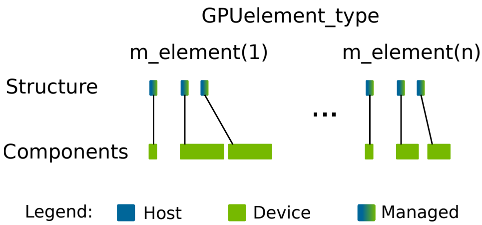
Two ways to address this.

Option1: device structure that points to the same device component memory:




```
! CPU valid version for use on the host:
type(GPUelement_type), pinned, allocatable :: GPUelement(:)
! Device structure that will point to same device components:
type(GPUelement_type), device, allocatable :: d_GPUelement(:)
 allocate ( GPUelement(Nelements) )
 allocate ( d_GPUelement(Nelements) )
! GPUelement%components(:) can be allocated in host code
! copy scalars and addresses of host struct to d struct :
cudaMemcpy(d_GPUelement, GPUelement, size(GPUelement))
```

BEST: ALLOCATE THE STRUCTURE AS MANAGED MEMORY:




```
! host and device valid structure (managed memory):
type(GPUelement_type), managed, allocatable :: m_GPUelement(:)

allocate ( m_GPUelement(Nelements) )
! m_GPUelement%components(:) can be allocated in host code
! they still live on the device
...
! can prefetch structure to device to avoid pagefault :
cudaMemPrefetchAsync(m_GPUelement,size(m_GPUelement),device=
      stream=0)
```

Nested Data Structures

Two Topics:

- How do you make them referencable on the device?
- **How do you efficiently move data into them?**

Efficiently Populating Nested Structures

A naive implementation for getting data structures populated on the GPU usually looks like this:

```
! Host data  structure  has been created and populated.
! GPU data structure  has also been allocated .

!  still  need to poplulate  the  values  from the  host  version  of  the  data  structure :
do id=1, Nelements
   GPUelement(id)%Nnodes = element(id)%Nnodes ! implicit cudaMemcpy
   GPUelement(id)%x = element(id)%x          !  implicit  cudaMemcpy
   GPUelement(id)%y = element(id)%y          !  implicit  cudaMemcpy
   GPUelement(id)%val = element(id)%val      !  implicit  cudaMemcpy
enddo
```

This becomes very slow when Nelements is large.

EFFICIENTLY POPULATING NESTED STRUCTURES

There are two ways to fix the naive approach,

1. BETTER: push from host with cudaMemcpyAsync calls instead of the numerous blocking calls done above,
2. BEST: pull the data from the GPU by populating the arrays from within a device kernel using zero copy.

Push with loop of Async calls

Issue copy calls to default stream which is asyncronous to the CPU:

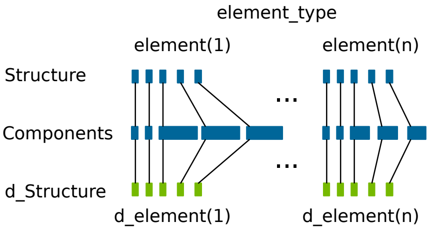
```
do id=1, Nelements
    GPUelement(id)%Nnodes = element(id)%Nnodes ! cpu to cpu copy
     istat =cudaMemcpyAsync(GPUelement(id)%x, element(id)%x, size(element(id)%x), stream=0)
     istat =cudaMemcpyAsync(GPUelement(id)%y, element(id)%y, size(element(id)%y), stream=0)
     istat =cudaMemcpyAsync(GPUelement(id)%val, element(id)%val, size(element(id)%val), stream=0)
enddo
```

Can do similar in OpenMP 4, with update nowait.

Host threading over id loop made no difference in performance.

PULLING FROM THE GPU

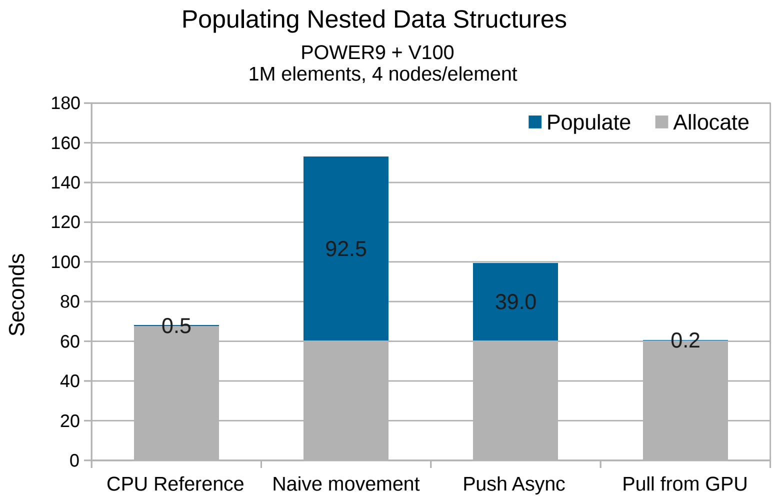
Set up a structure to reference pinned host components from the device (zero copy of structure components):




```
! to use element on the device, we have to make a device valid copy called d_element:
istat =cudaMemcpyAsync(d_element, element, size(element), 0)

! Now we can use these in a CUDA kernel to zero copy
! the component data from d_element into d_GPUelement
call set_elements_kernel <<<blocks,threads>>>(d_GPUelement,d_element,Nelements)
```

Launch a kernel to have GPU threads pull data from structures on the host into structures on the device:

```
attributes(global) subroutine set_elements_kernel(GPUelement,element, Nelements)
  implicit none
  ! kernel that uses zero copy to popluate the GPUelement structure.
  type(GPUelement_type), device, intent(inout) :: d_GPUelement(:) ! members are device
  type(element_type), device, intent(in) :: p_element(:)  ! members are pinned host
  integer, value, intent(in) :: Nelements
  integer :: id, Nnodes, node

  do id=blockIdx%x,Nelements, gridDim%x
     Nnodes = p_element(id)%Nnodes
     d_GPUelement(id)%Nnodes = Nnodes
     do node = threadIdx%x, Nnodes, blockDim%x
        d_GPUelement(id)%x(node) = p_element(id)%x(node)
        d_GPUelement(id)%y(node) = p_element(id)%y(node)
        d_GPUelement(id)%val(node) = p_element(id)%val(node)
     enddo
  enddo

end subroutine set_elements_kernel
```



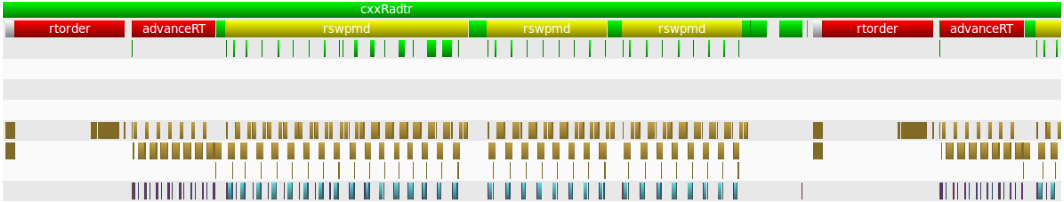
462x speedup when pulling from the GPU!

Closing Remarks

nvtx marker module available for Fortran

- Easily mark regions of host code for viewing in the Nvidia Visual Profiler.
- Works with CUDA, OpenMP, and OpenACC.
- Newly supports non-nested marked regions.
- Very helpful to understand flow of your application.

Availabe at https://github.com/dappelha/gpu-tips/nvtx



Conclusions

- Pinning memory is important, even when using directives
- Managed memory makes nested data structure book keeping easier.
- Still important to efficiently populate data structures.
- Zero-copy populating from the device is the fastest method (462x).

Various example codes are availabe at
`https://github.com/dappelha/gpu-tips`

Questions?

David Appelhans - dappelh@us.ibm.com