Handling Asynchronicity

Exploiting overlap

- **If individual kernels are performing well**
  - Can you start overlapping different computational tasks?
    - kernels on the GPU
    - data transfers to/from the GPU
    - maybe even separate computation on the CPU

- **You can do this using the async features**
  - OpenACC: stream handles; very similar to streams in CUDA
  - OpenMP: data dependencies; same as OpenMP tasks

Asynchronicity with Nvidia GPUs

- **accelerator operations are launched from the CPU**
- **they then execute asynchronously**
  - control returns immediately to the host
  - host must then wait or test for completion
    - automatically (e.g. handled by compiler)
    - manually (e.g. via directives or API calls)
  - applies to:
    - computational kernels
    - data transfers: update directives
    - plus some other directives

Automatic synchronisation

- **By default, the compiler will handle synchronisation:**
  - may be conservative, and wait for every operation to complete; or
  - may be smarter, and reduce number of sync. points (consistent with correctness)

- **CCE-specific options:**
  - control behaviour with compiler option -hacc_model=auto_async_*
    - auto_async_none :
      - waits for every operation to separately complete
      - useful for debugging and generating profiles
      - but it may skew performance
    - auto_async_kernels :
      - may allow some computational kernels to overlap
      - the default behaviour
    - auto_async_all :
      - may allow kernels and data transfers to overlap
      - try this as a performance-tuning option

A stream of tasks in OpenMP 4.0 (and 4.5)

- **Can simply add nowait clause to kernels, updates**
  - Over-rides automatic synchronisation

- **Operations:**
  - guaranteed to execute sequentially in the order they were launched
  - not guaranteed to execute immediately after each other; there could be delays
  - this is known as a "stream" of tasks

- **Synchronisation**
  - taskwait or barrier directives ensure all asynchronous operations have completed

- **If you've used CUDA streams, these concepts should be very familiar**

Streams of tasks in OpenMP 4.5

- **Asynchronicity handled in the same way as tasks (4.0)**
  - Each kernel or set of data transfers is a task
- **depend clause gives much greater control than nowait**
  - Over-rides automatic synchronisation
  - Allows user to specify dependencies between tasks
  - These control:
    - order of execution
    - whether tasks can overlap or not

Dependency-types

- **Dependencies are expressed as data dependencies**
  - depend(dependency-type:variable)
- **Dependency types are:**
  - in:
    - in:a task cannot start until all tasks with out:a or inout:a have finished
  - out:
    - out:a task must complete before any in:a or inout:a tasks can start
  - inout: combination of in and out behaviour for same variable

Dependency variables

- **in/out/inout dependency types**
  - Simply used to define a dependency tree
  - Do not imply any data movement to/from the accelerator
  - Do not express how the variables is used in the task
    - in:a does not mean a is used in a read-only fashion in the task
      - a could be used in a write-only or read-write fashion, or
      - a might not be referenced at all in the task

OpenMP depend first example

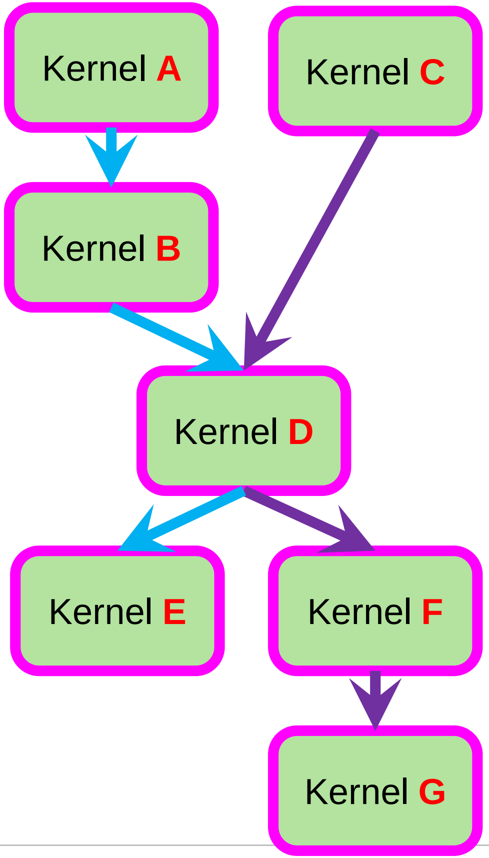
- **a simple pipeline:**
  - processing an array, slice by slice
    - assume can be processed independently
  - each slice requires 3 tasks:
    - copy data to GPU,
    - process on GPU,
    - bring back to CPU
  - which must execute sequentially
  - but we can overlap different slices

```
REAL :: a(Nvec,Nchunk),b(Nvec,Nchunk)

!$omp target data map(alloc:a,b)
DO j = 1,Nchunks
!$omp target update to(a(:,j)) &
      depend(out:a(:,j))

!$omp target teams distribute &
      depend(inout:a(:,j))
  DO i = 1,Nvec
    b(i,j) = <function of a(:,j)>
  ENDDO
!$omp end target teams distribute

!$omp target update from(b(:,j)) &
      depend(in:a(:,j))

ENDDO
!$omp taskwait
!$omp end target data
```

- **Expect to see overlap of three streams at once**
    - one sending to the device; one processing the slice; one sending to host

High-level example with OpenMP 4.5

```
!$omp target depend(out:x)
<Kernel A>
!$omp target depend(inout:x)
<Kernel B>

 !$omp target depend(out:y)
 <Kernel C>

!$omp target depend(inout:x) &
             depend(inout:y)
<Kernel D>

 !$omp target depend(in:x)
 <Kernel E>

 !$omp target depend(inout:y)
 <Kernel F>

 !$omp target depend(in:y)
 <Kernel G>

 !$omp taskwait ! all completed
```



Synchronising subsets of tasks

- **Dependencies ensure that tasks execute in order**
- **Can also globally synchronise**
  - taskwait and barrier directives
- **May also want to add CPU task into stream, e.g.**
  - kernel to pack buffer on accelerator
  - update to move buffer back to host

A stream of tasks in OpenACC

- **Add async clause to make operation asynchronous**
  - kernel or data transfer
- **Operations:**
  - guaranteed to execute sequentially in the order they were launched
  - not guaranteed to execute immediately after each other; there could be delays
  - this is known as a "stream" of tasks
- **Synchronisation**
  - wait directive ensures all asynchronous operations have completed
  - can use an API call to do same thing (or to test for completion)
- **If you've used CUDA streams, these concepts should be very familiar**

Multiple streams of tasks in OpenACC

- **The async clause can take a handle**
  - handle is an argument that is a positive- or zero-valued integer
  - each handle value is a separate stream of tasks
- **Tasks within a given stream**
  - guaranteed to execute sequentially in order they were issued
- **Tasks in different streams**
  - may overlap or serialise, as the hardware and runtime allows
  - operations in different streams should be independent or we have a race condition
- **Synchronisation**
  - wait directive ensures all streams of tasks have completed
  - wait(handle) directive ensures just the specified stream of tasks has completed
  - can use an API call to do same thing (or to test for completion)
- **If you've used CUDA streams, these concepts should be very familiar**

OpenACC async first example

- **a simple pipeline:**
  - processing an array, slice by slice
  - assume slices can be processed independently
  - each slice requires 3 tasks:
    - copy data to GPU,
    - process on GPU,
    - bring back to CPU
  - which must execute sequentially
  - but we can overlap different slices
  - Use a different stream for each slice
    - use slice number as stream handle
    - don't worry if number gets too large

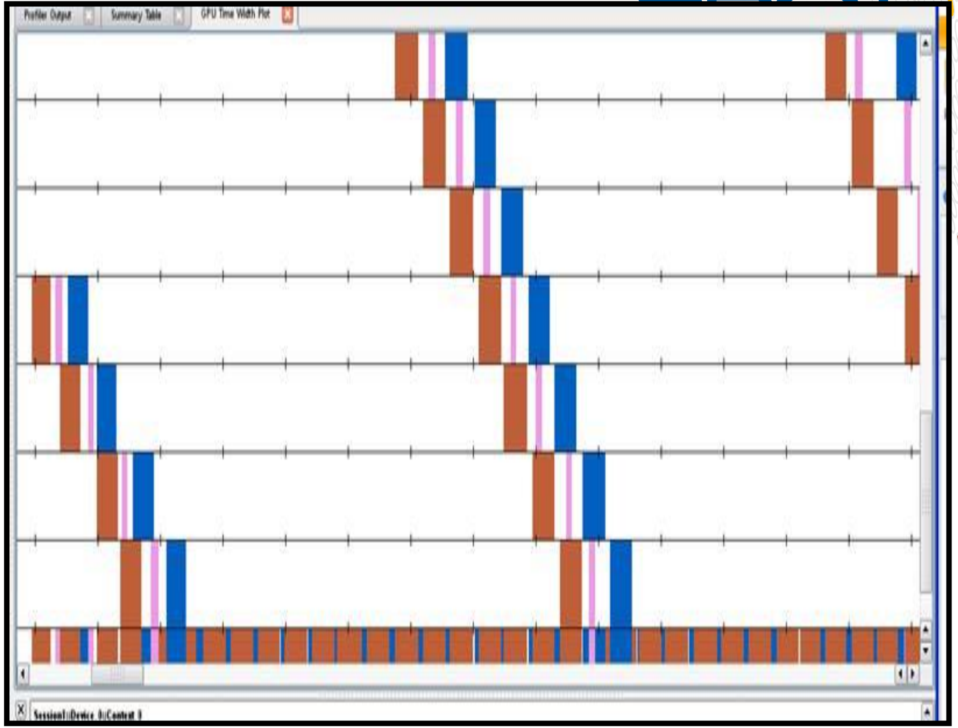
```
REAL :: a(Nvec,Nchunk),b(Nvec,Nchunk)

!$acc data create(a,b)
DO j = 1,Nchunks
!$acc update device(a(:,j)) async(j)

!$acc parallel loop async(j)
  DO i = 1,Nvec
    b(i,j) = <function of a(:,j)>
  ENDDO

!$acc update host(b(:,j)) async(j)

ENDDO
!$acc wait
!$acc end data
```

- **Expect to see overlap of three streams at once**
  - one sending to the device; one processing the slice; one sending to host

OpenACC async results

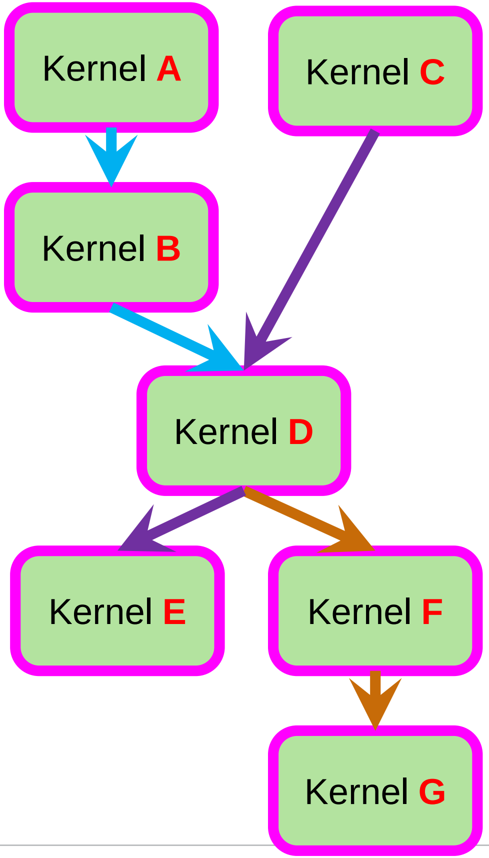
- **Execution times:**
  - CPU: 3.76s
  - OpenACC: 1.10s
  - OpenACC, async: 0.34s

- **NVIDIA Visual profiler**
  - only 7 of the 16 streams shown
  - collapsed view at bottom



High-level example with OpenACC

```
!$acc parallel loop async(stream1)
<Kernel A>
!$acc parallel loop async(stream1)
<Kernel B>
 !$acc parallel loop async(stream2)
 <Kernel C>
!$acc parallel loop async(stream3) &
!$acc      wait(stream1,stream2)
<Kernel D>
!$acc parallel loop async(stream4) &
!$acc      wait(stream3)
<Kernel E>
!$acc parallel loop async(stream5) &
!$acc      wait(stream3)
<Kernel F>

!$acc parallel loop async(stream5)
<Kernel G>

!$acc wait ! ensures all completed
```



What's missing in OpenACC and OpenMP?

- **Deep copy is the elephant in the room**
- **Hierarchical data structures with pointers**
  - C++ objects; C structs; Fortran derived types
  - On CPU, pointers point to CPU memory addresses
  - When "map" to GPU, pointers still point to CPU addresses
    - main impact: at best, inefficient; usually, just broken
  - User needs to explicitly remap all the pointers
    - this is not a very satisfactory solution
- **CCE -hacc_model=deep_copy helps for Fortran only**
- **Proper solution deferred to OpenMP 5.0 (due Nov.17)**
  - and OpenACC 3.0 (due ?)