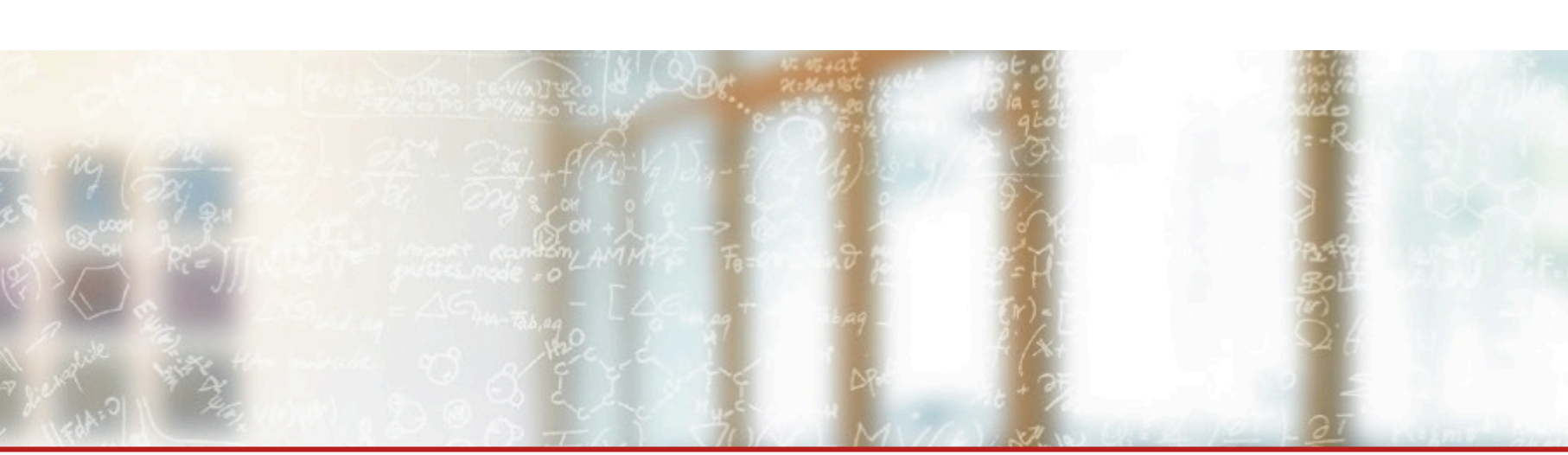


Introduction to OpenACC

Markus Wetzstein, CSCS

June 19, 2015

GPU programming models

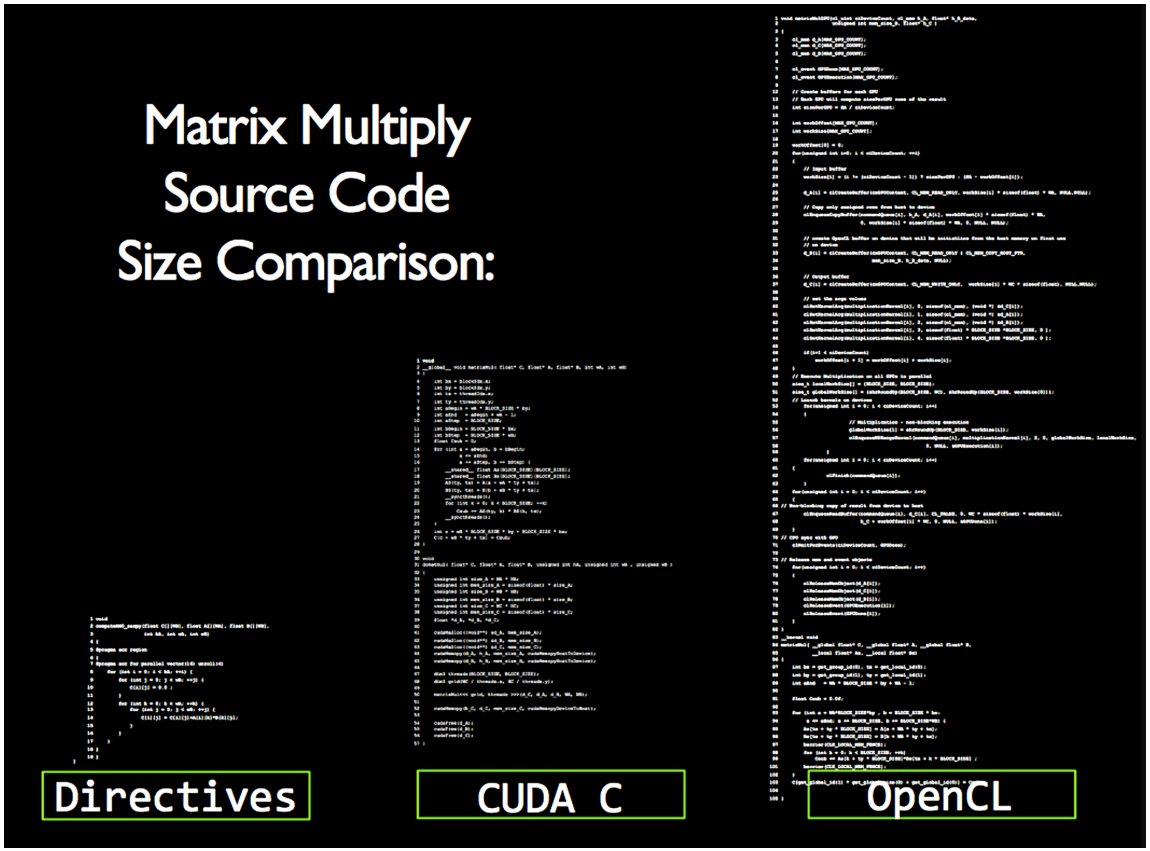
APIs currently available:

- CUDA
  - NVIDIA product, best performance, low portability
- OpenCL
  - Open standard, API similar to CUDA, high portability
- **OpenACC**
  - **Directive based, our focus in this session**
- Libraries
  - MAGMA, cuFFT, Thrust, CULA, cuBLAS

Why OpenACC?

CUDA and OpenCL are quite low-level and closely coupled to GPU functionality, but:

- Potentially high initial investment to port existing code to GPUs
- User needs to rewrite kernels in ‘specialist’ language:
  - Hard to write and debug
  - Hard to port to new accelerator
- Maintenance problem if both CPU & GPU version of code need to be kept
  - Multiple versions of kernels in codebase
  - Hard to add new functionality



OpenACC: http://www.openacc.org/

Accelerator programming API standard

- Allows parallel programmers to provide simple hints, known as "directives" to the compiler, identifying which areas of code to accelerate
- Aimed at incremental development of accelerator code
- Supported by various vendors
  - Cray
  - PGI
  - gcc: experimental feature in gcc v5.1, full OpenACC 2.0a compliance in future release
  - Pathscale Enzo

OpenACC: directive-based programming

+ Based on original source code (Fortran, C, C++)
  + Easier to maintain/port/extend code
  + Users with OpenMP experience find it a familiar programming model
  + Compiler handles repetitive coding (cudaMalloc, cudaMemcpy...)
  + Compiler handles default scheduling; user tunes only where needed

– Possible performance sacrifice

OpenACC vs OpenMP 4.0

- in the OpenMP 4.0 standard, accelerator extensions have been defined
- so why use OpenACC ?
- historically, OpenACC was created to get a directive based accelerator programming model out and running before OpenMP might provide it (driven by Cray, PGI)
- OpenACC compiler implementations are often more mature than OpenMP4.0 accelerator implementations
- some support for more complicated data structures and also some operations is simply not there yet in OpenMP 4.0

Execution Model

- CPU is the ‘driver’
- compute intensive regions offloaded to accelerators
- accelerators execute `parallel` and `kernels` regions
- the host is responsible for:
  - Allocation de-allocation of memory in accelerator
  - Data transfers (from-to the accelerator)
  - Sending the code to the accelerator
  - Waiting for completion
  - Queue sequences of operations executed by the device
- work is scheduled in gangs, workers, vectors (mapping to CUDA blocks, warps, threads)

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

OpenACC directives

OpenACC directives

- Directives facilitate code development for accelerators
- Provide the functionality to:
  - Manage data transfers between host (CPU) and accelerator
  - Manage the work between the accelerator and host.
  - Manage computations (loops) on accelerators
  - Tune code for performance

Basic directives format

Modify original source code with directives

- Non-executable statements (comments, pragmas)
- Sentinel: **acc**
  - **C/C++**:
    - preceded by `#pragma`
    - Structured block {...} avoids need for `end` directives
  - **Fortran**:
    - Preceded by `!$` (f90 and later) or `c$` (f77)
    - Usually paired with `!$acc end *` directive
    - Directives can be capitalized

```
/* C/C++ example */
#pragma acc *
{structured block}
```

```
! Fortran example
!$acc *
<structured block>
!$acc end *
```

- Continuation to extra lines allowed
  - **C/C++:** \ (at end of line to be continued)
  - **Fortran:**
    - Fixed form: `c$acc&` or `!$acc&` on continuation line
    - Free form: **&** at end of line to be continued
      - continuation lines can start with either `!$acc` or `!$acc&`

Conditional compilation

- In theory, OpenACC code should be identical to CPU
  - only difference are the directives (i.e. comments)
  - can be ignored by non-accelerating compiler
  - OpenACC compilers have flags to compile without OpenACC
    e.g. Cray compiler (CCE): `-hnoacc` suppresses compilation
- In practice, the final code will be (possibly highly) different
  - Substantial code refactoring is needed:
    - usually for performance reasons
    - usually better OpenACC code is better CPU code
- Sometimes necessary to have different code sniplets for CPU/GPU
  - OpenACC defines preprocessor macro: `_OPENACC`

```
#ifdef _OPENACC
…
#else
…
#endif
```

A first example

```
program example
  integer, parameter:: n=100
  real, dimension(n):: a,b,c
  …
!$acc parallel copyin(b,c) copyout(a) &
!$acc& private(i)
!$acc loop
  do i=1,n
     a(i)=b(i)+c(i)*i
  enddo
!$acc end parallel
  …
end program
```

Note:

most data clauses here are redundant because of compiler defaults

One kernel:

- define a parallel region for the GPU, workshare the loop among threads (`parallel, loop`)

Two types of data clauses:

- define access for the threads to the data (`private`)
- define data transfer between CPU/GPU (`copyin, copyout`)

OpenACC main classes of directives

- accelerator `parallel` region / `kernels` directives
- `loop` directive
- data scoping & transfer directives
- synchronization directives
- `cache` directive
- `atomic` directive

Some directives can be fused together, e.g.

```
#pragma acc parallel loop
for(…){
…
}
```

instead of

```
#pragma acc parallel
{
#pragma acc loop
for(…){
…
}
}
```

`parallel` / `kernels` – the OpenACC workhorses

- Parallel region (`!$acc parallel [clause]`):
  part of the code executed in parallel on the accelerator keeping gangs, workers and vector constant
- Kernel (`!$acc kernels [clause]`):
  region of a program that is to be compiled into a sequence of kernels for execution on the accelerator. When program encounters a kernels construct, it will launch sequence of kernels in order on the device. Number and configuration of gangs of workers and vector length may be different for each kernel.

Second example – `parallel` vs `kernels`

```
program example_parallel
  …
!$acc parallel [data clauses]
!$acc loop
  do i=1,n
    …
  enddo
!$acc loop
  do j=1,m
    …
  enddo
!$acc end parallel
…
end program example_parallel
```

- one constant set of thread blocks for both loops

```
program example_kernels
  …
!$acc kernels [data clauses]
  do i=1,n
    …
  enddo
  do j=1,m
    …
  enddo
!$acc end kernels
…
end program example_kernels
```

- possibly two sets of thread blocks for the loops

Visually almost the same, but some subtle differences exist

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Data scoping in OpenACC

Data scoping

- In a serial code (or pure MPI), there are no complications
- In a thread-parallel code (OpenACC, OpenMP etc.) things are more complicated:
  - Some data will be the same for each thread (e.g. the main data array)
    - The threads can (and usually should) share a single copy of this data
  - Some data will be different (e.g. loop index values)
    - Each thread will need it's own private copy of this data
- Data scoping arranges this. It is done:
  - automatically (by the compiler) or explicitly (by the programmer)

- If the data scoping is incorrect, we get:
  - incorrect (and inconsistent) answers ("race conditions"), and/or
  - a memory footprint that is too large to run

Understanding data scoping

- Data scoping ensures the right answer
  - We want the same answer when executing in parallel as when serially
- Declare variables in parallel region to be `shared` or `private`
  - `shared`
    - all loop iterations process the same version of the variable
    - variable could be a scalar or an array
    - `a` and `b` are `shared` arrays in this example
  - `private`
    - each loop iteration uses the variable separately
    - again, variable could be a scalar or an array
    - `t` is a `private` scalar in this example
    - loop index variables (like `i`) are also private
  - `firstprivate`: a variation on `private`
    - each thread's copy set to initial value
    - loop limits (like `N`) should be `firstprivate`

```
for (i=0; i<N; i++) {
   t = a[i];
   t++;
   b[i] = 2*t;
}
```

Data scoping (2)

- In OpenMP, we have the following data clauses
  - `shared, private, firstprivate, lastprivate`
- In OpenACC
  - `private, firstprivate` are just the same (`lastprivate` doesn't exist)
- `Shared` variables are more complicated in OpenACC because we also need to think about data movements to/from GPU

OpenACC `parallel` regions:

- scalars and loop index variables are `private` by default
- arrays are `shared` by default
  - the compiler chooses which shared-type: `copyin, copyout`, etc.
- explicit data clauses over-ride automatic scoping decisions
- you can also add the `default(none)` clause
  - you have to do everything explicitly (or you get a compiler error)

GPU data management

When creating an accelerated region, data management can be explicitly specified through the following data clauses

- a comma-separated collection of variable names, array names, or subarray specifications. Compiler allocates and manage a copy of variable or array in device memory, creating a visible device copy of variable or array.
- **Data clauses:** **`copy`**

  `copyin`

  `copyout`

  `create`

  `present`

  `present_or_copy` or short **`pcopy`**

  `present_or_copyin` or short **`pcopyin`**

  `present_or_copyout` or short **`pcopyout`**

  `present_or_create` or short **`pcreate`**

  `deviceptr`

Example:

```
#pragma acc parallel loop \
copyin(a[start:len], b[start:len]),\
copyout(c[start:len])
---------------
!$acc parallel loop &
!$acc& copyin(b,c) copyout(a)
```

Data transfer clauses explained

- `copy`: `copyin` and `copyout`
- `copyin`: transfer CPU ➔ GPU at entry of accelerator region
- `copyout`: transfer CPU ➔ GPU at exit of accelerator region
- `present`: object exists in accelerator memory already (error if not!)
- `create`: allocate object in accelerator memory without transfer to/from CPU
- ….plus the various combinations with `present_or_X`
- `deviceptr`: object is in accelerator memory already, so no need to copy or transfer CPU -> GPU (only few use cases)

Differences between Fortran and C/C++

- in Fortran, the pointer carries the information of the shape and dimensions of the array
- in C, a pointer is a memory address, without this information

Example:

```
#pragma acc parallel loop \
copyin(a[start:len], b[start:len]),\
copyout(c[start:len])
---------------
!$acc parallel loop &
!$acc& copyin(b,c) copyout(a)
```

This means:

- in Fortran, if we specify `copyin(b,c)` the compiler knows everything
- in C, it knows where a transfer starts, but not where it ends and what the dimensions are
- In most cases, you'll need to specify the dimensions in C, using a `[start:len]` pair

Data regions

Data movement example (1)

```
program main
  integer :: a(N)
  <stuff>
!$acc parallel loop copyout(a)
  do i = 1,N
   a(i) = i
  enddo
!$acc end parallel loop
!$acc parallel loop copy(a)
  do i = 1,N
   a(i) = 2*a(i)
  enddo
!$acc end parallel loop
  <stuff>
end program main
```

- first loop initializes array `a`
- second loop does further computation
- only data transfer needed: `copyout` after the second loop
- BUT: we have two additional, unneccessary transfers

use OpenACC data regions

OpenACC data regions

- Data regions allow data to remain on the accelerator
  - e.g. for processing by multiple accelerator kernels
  - specified arrays only move at start/end of data region

- Data regions only label a region of code
  - they do not define or start any sort of parallel execution
  - just specify GPU memory allocation and data transfers
  - can contain host code, nested data regions and/or device kernels

- Be careful:
  - Inside data region we have two copies of each of the specified arrays
  - These only synchronise at the start/end of the data region
    - and only following the directions of the explicit data clauses
  - Otherwise, you have two separate arrays in two separate memory spaces

Defining OpenACC data regions

- Two ways to define data regions:
  - Structured data regions:
    - Fortran: `!$acc data [data-clauses] ... !$acc end data`
    - C/C++: `#pragma acc data [date-clauses]` {...}
  - Unstructured data regions (new in OpenACC 2.0):
    - Fortran:
      ```
      !$acc enter data [data-clauses]
      ...
      !$acc exit data [data-clauses]
      ```
    - C/C++:
      ```
      #pragma acc enter data [data-clauses]
      ...
      #pragma acc exit data [data-clauses]
      ```
- For most "procedural code", use structured data regions
- Unstructured data regions
  - Useful for more "Object Oriented" coding styles, e.g.
    - Separate constructor/destructor methods in C++
    - Separate subroutines for malloc (or allocate) and free (or deallocate)
- A data region with no data clauses is "like a broken pencil"
  - pointless (that is, redundant)

Data region example

```
program main
  integer :: a(N)
!$acc data copyout(a)
  <stuff>
!$acc parallel loop
  do i = 1,N
   a(i) = i
  enddo
!$acc end parallel loop
!$acc parallel loop
  do i = 1,N
   a(i) = 2*a(i)
  enddo
!$acc end parallel loop
!$acc end data
  <stuff>
end program main
```

only allocation

here is the transfer

Data region allows us to:

- separate the data transfers from the kernels
- only do the necessary transfer

Sharing GPU data between subroutines

```
program main
  integer :: a(N)
!$acc data copyout(a)
  <stuff>
!$acc parallel loop
  do i = 1,N
   a(i) = i
  enddo
!$acc end parallel loop
  call double_array(a)
!$acc end data
  <stuff>
end program main
```

```
subroutine double_array(b)
  integer :: b(N)
!$acc parallel loop present(b)
  do i = 1,N
   b(i) = double_scalar(b(i))
  enddo
!$acc end parallel loop
end subroutine double_array
```

```
subroutine double_scalar(c)
  integer :: c
  double_scalar = 2*c
end subroutine double_scalar
```

- `present` clause uses GPU version of b without data copy
  - Original calltree structure of program can be preserved
- One kernel is now in subroutine (maybe in separate file)
  - OpenACC 2.0: compilers support nested parallelism, no inlining needed

Data transfers within a data region

```
!$acc update [host | device] (var-list)
```

- Common use case:
  - data is needed back in host memory before the data region has ended
  - or new data in host memory needs to be transferred to accelerator
  - e.g. due to MPI communications

```
int main(void) {
  double a[100];
 <stuff>
#pragma acc data copyout(a)
 {
  <some kernel>
#pragma acc update host(a[0:2],
a[98:2])
  <use mpi to exchange boundary>
#pragma acc update device(a[0:2],
a[98:2])
 <some more kernels>
 }
 <stuff>
}
```

Note: some OpenACC implementations allow MPI directly from GPU to GPU memory, e.g. Cray on XC series of machines

Data scoping recap

- **`parallel`** regions:
  - scalars and loop index variables are **`private`** by default
  - arrays are **`shared`** by default
    - the compiler chooses which `shared`-type: `copyin, copyout`, etc.
  - explicit `data` clauses over-ride automatic scoping decisions
- **`data`** regions:
  - only `shared`-type scoping clauses are allowed
  - there is **NO** default/automatic scoping
  - un-scoped variables on data regions
    - will be scoped at each of the enclosed `parallel` region automatically, unless the programmer does this explicitly
    - this probably leads to unwanted data-movements for large arrays
  - using data region scoping in enclosed `parallel` regions:
    - same routine: omit scoping clauses on enclosed `parallel` directives
    - different routine: use `present` clause on enclosed `parallel` directives

Summary (… so far)

- Compute regions
  - created using `parallel` or `kernels` directives
- Data regions
  - created using `data` or `enter/exit data` directives
- Data clauses are applied to:
  - accelerate loopnests: `parallel` and `kernels` directives
    - here they over-ride relevant parts of the automatic compiler analysis
    - you can switch off all automatic scoping with `default(none)`
  - data regions: `data` directive and `enter/exit data`
    - Note there is no automatic scoping in data regions (arrays or scalars)
  - Shared clauses (`copy, copyin, copyout, create`)
    - supply list of scalars, arrays (or array sections)
  - Private clauses (`private, firstprivate, reduction`)
    - only apply to accelerated loopnests (`parallel` and `kernels` directives)
  - `present` clause (used for nested data/compute regions)

Exercise: axpy with OpenACC

Open the `axpy.cpp` or `axpy.f90` example

- put the axpy kernel on the GPU using OpenACC
  - first, limit the directives to the axpy function
- once you've verified the results are ok:
  - run the code for a set of array sizes that you have already timed with CUDA
  - compare the timings
- implement the data transfers with a data region, outside of the axpy kernel and its timer
  - add timers for the transfers themselves
  - rerun for previous array sizes
  - compare:
    - what fraction of the GPU time is in the transfers?
    - how does the OpenACC time for the kernel compare to the CUDA time?

Some maybe useful info and tricks

- the makefile is written such that the Cray compiler is used
- so you need to have the module `PrgEnv-cray` (which is the default) loaded PLUS the module `craype-accel-nvidia35` to get GPU code
- with Cray, you can specify `–rm` (Fortran) / `-h list=m` (C) to get a listing file `*.lst` to get a compiler listing with annotations which kernels have been created, which transfers, etc.
- environment variable `CRAY_ACC_DEBUG` can be used at runtime to provide details about which data (with sizes) gets transferred where, which kernels get started and which launch configurations are used for them
  typically `CRAY_ACC_DEBUG=2` gives all the info you might want

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced topics in OpenACC

Motivation and Outlook

there surely has to be more to OpenACC…???

…yes, indeed:

- tuning workshare constructs
- reduction operations
- asynchronous operations
- interoperability with CUDA and libraries

I have an existing OpenMP code, how easy is it to port to OpenACC?

…fairly easy, BUT:

important differences between OpenMP and OpenACC (although it all looks very similar)

Parallel region with multiple workshared loops ?

- often used construct in OpenMP
- thread creation/destruction @ begin/end of parallel region
- loops are workshared
- code between loops is executed redundantly on each thread

what about OpenACC ?

```
!$omp parallel shared(…) &
!$omp& private(…) […]

!$omp do
do i=istart,iend
   …
enddo
[some code, not workshared]
!$omp do
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$omp end parallel
```

Use of parallel and loop

- OpenACC parallel region can span several loops

  BUT

- OpenACC ≠ OpenMP although visually very similar
- each loop executed by the same sets of threads
- no synchronization between these and none with code in between !
- no global barrier mechanism inside parallel region
- very easy to create race conditions

```
!$acc parallel shared(…) &
!$acc& private(…) […]

!$acc loop
do i=istart,iend
   …
enddo
[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$acc end parallel
```

Use of parallel and loop (2)

```
!$acc parallel shared(…)
private(…) […]

!$acc loop
do i=istart,iend
   …
enddo

[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo

!$acc end parallel
```

```
!$omp parallel shared(…) &
!$omp& private(…) […]

!$omp do
do i=istart,iend
   …
enddo
!$omp end do nowait
[some code, not workshared]
!$omp do
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$omp end do nowait
!$omp end parallel
```

Use of parallel and loop: recommendations

- only use composite `parallel loop` at first
  ➔ get correct code
- understand `loop` by itself as corresponding to having an implicit `nowait` clause in OpenMP (without the option to have a `wait`!)
- carefully separate directives as a later performance tuning step:
  - only if you're sure the loops can be independent kernels and have no race conditions !
  - consider explicitly using `async` clause instead
    ➔ slightly more complicated, but enhanced code clarity

```
!$acc parallel shared(…) &
!$acc& private(…) […]

!$acc loop
do i=istart,iend
   a(i)=b(i)*const + c(i)
enddo
[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      x(j,k)=y(j,k)+z(j,k)
   enddo
enddo
!$acc end parallel
```

- ✓ loops independent!
- ✓ potentially slightly faster than having two separate `parallel loop` clauses

Kernels vs parallel regions

- what is the **difference** between **`kernels`** and **`parallel`** ?
  - very similar usage
  - have different historic origins (`kernels` ➔PGI, `parallel` ➔Cray)
  - OpenACC standard not very helpful to understand when to use what

```
#pragma acc parallel loop […]
for(i=istart ;i<=iend; i++) {
   …
}
#pragma acc kernels loop […]
for(i=istart ;i<=iend; i++) {
   …
}
```

in common:

- both define a region to be accelerated

differences:

- different levels of obligation to compiler

parallel	**kernels**
1 kernel	≥1 kernel(s)
must be accelerated	can be accelerated
tuning clauses	no tuning clauses

Kernels vs parallel regions (2)

- compiler will automatically analyze all loops inside `kernels`
- BUT: with kernels, first loop is guaranteed to have finished before second starts

```
#pragma acc kernels […]
{
for(i=istart ;i<iend; i++) {
   a[i]=b[i]*c[i];
}
for(i=istart+1 ;i<iend-1; i++) {
   d[i]= 0.5*(a[i-1]+a[i+1]);
}
} //acc kernels
```

What to use…?

- `parallel` offers greater control
- `kernels` maybe better to initially explore parallelism

suggestion:

- don't mix them unless you're really aware of the subtle differences

Tuning parallel execution

```
[parallel] loop [gang] [worker] [vector]
```

- parallel execution structured into hierarchy:
  `gang` → `worker` → `vector`
- code is executed in parallel with current level of parallelism until a new level is opened
  (`gang`: redundant execution)
- optional, compilers define defaults (possibly using heuristics)
- only allowed on `loop` directive

```
!$acc parallel loop […] gang
do i=istart,iend
!$acc loop worker
   do j=jstart,jend
!$acc loop vector
      do k=kstart,kend
         …
      enddo
   enddo
enddo
!$acc end parallel
```

OpenACC	CUDA
`gang`	threadblock
`worker`	warp of threads
`vector`	threads

when to use it, and why?

THIS is not a very efficient use case !

Tuning parallel execution (2)

Explicitly using multiple levels of parallelism:

- loop iterations must be data independent (except `reduction`s)
- usage: indirect indexing, …
- `worker` and `vector` loops have an implied barrier at end of loop

`loop [collapse(Nlevels)]`

- specifies how many tightly nested loops are associated with a `loop` construct
- without `collapse` a `loop` construct only affects the immediately following loop

```
!$acc parallel loop […] gang
do i=istart,iend
   inew=index_list(i)
!$acc loop worker
   do j=jstart,jend
      jnew=index_list2(j)
!$acc loop vector
      do k=kstart,kend
         a(k,j,i)=b(k,jnew)+c(k,inew)
      enddo
   enddo
enddo
!$acc end parallel
```

```
!$acc parallel loop […] collapse(3) &
!$acc& gang worker vector
do i=istart,iend
   do j=jstart,jend
     do k=kstart,kend
        a(k,j,i)=b(k,j)+c(k,i)
      enddo
   enddo
enddo
!$acc end parallel
```

Tuning parallel execution (3)

```
parallel [num_gangs(N1)] [num_workers(N2)] [vector_length(N3)]
```

- `num_gangs`: nr. of gangs to use for parallel region (integer)
- `num_workers`: nr. of workers to use for `worker` loops (integer)
- `vector_length`: nr. of threads to use for `vector` loop (integer)

- binds to `parallel`, not `loop`
- if omitted, compiler chooses itself
- `vector_length`: compiler might allow only certain values
  e.g. Cray: 1, 64, 128 (default), 256, 512, 1024
- Cray only allows:
  either `num_workers` (fixes `vector_length=32`)
  or `vector_length` (fixes `num_workers=vector_length/32`)

Tuning parallel execution (4)

Some suggestions:

- explicitly using `worker` as separate level often not very useful (in current implementations)
- tightly nested loops: try if `collapse` improves performance
- tuning `num_gangs|num_workers|vector_length`
  - is time consuming
  - optimal choice depends on actual loop
  - focus on expensive loops
- to debug kernel by running single thread, use:
  `#pragma acc parallel num_gangs(1) vector_length(1)`

Reduction operations

```
[parallel | loop] reduction(operator:variable-list)
```

- OpenACC reductions very similar to OpenMP
- `reduction` only allowed for scalars
  arrays: rewrite to use temporary scalars inside loop nest for reduction
- reduction variable is `private` to each thread
- combine result over all threads
  e.g. sum, max, min, logical and
- careful: reduction over `gangs` only done at end of parallel construct !

```
#pragma acc parallel loop
        reduction(+:t)
for(i=istart;i<=iend;i++) {
   t=t + a[i] – b[i];
}
```

Reduction operations (2)

C / C++		Fortran	
operator	initialization	operator	initialization
+	0	+	0
*	1	*	1
max	least	max	least
min	largest	min	largest
&	~0	iand	all bits on
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

initialization automatically by compiler (based on operation)

```
#pragma acc parallel […] {
[some code]
#pragma acc loop reduction(+:t)
        gang vector collapse(2)
for(i=istart;i<=iend;i++) {
   for(j=jstart;j<=jend;j++) {
      t = t + a[j,i] – b[j,i];
   }
}
// reduction of t is INCOMPLETE
// using t here=race condition
} // end acc parallel

// using t here is OK
```

be careful with reductions over gangs before exiting parallel region

Exercise: dot product with OpenACC

Write a dot product kernel

- reuse the `axpy` example:
  - it has two vectors and a scalar as arguments, so that works
  - you'll need to adjust the transfers and the kernel itself
- compare the amount of work it took to that of the CUDA version yesterday

Use of data on GPU in libraries / CUDA

```
!$acc host_data use_device(var-list)
```

- how to pass a pointer to memory on the GPU to a library, or to a CUDA kernel? E.g. to:
  - use third party GPU library (e.g. Cray libsci_acc, cuBLAS, cuFFT, …) to process data already held on device
  - use optimized CUDA kernel to process data already held on device
  - use optimized MPI library to transfer data across nodes directly between the GPU memories
- `host_data` makes a pointer on the device available on the host
- nested inside `data` region which put *`var-list`* on the GPU

Interoperability with CUDA

```
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();
}
```

- **Call CUDA-C wrapper (compiled with nvcc; linked with normal compiler)**
  - must include cudaThreadSynchronize()
  - Before: so asynchronous accelerator kernels definitely finished
  - After: so CUDA kernel definitely finished before we return to the OpenACC
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

Asynchronous operations

- GPUs have more than one queue (CUDA: stream) into which operations can be entered
  - in hardware: Nvidia Fermi 16, Kepler 32 (with better hardware to overlap those), nr. of logical queues even much higher
- operations in different queues can be executed concurrently
- CPU can continue execution immediately after adding an operation to a queue ➔ no need to wait for completion of actual operation
- potential performance gains from:
  - overlapping data transfer with computation on GPU
  - overlapping data transfer with computation on CPU
  - expose more parallelism to the GPU (e.g. multiple kernels and data transfers at the same time)

List of asynchronous clauses / directives

```
wait[(handle-list)] [async(handle)]
```

- synchronisation directive
- *handle*: non-negative integer denoting the queue
- *handle-list*: list of handles, can only be used with `wait`
- `wait`: wait until all asynchronous operations have completed
- `wait(`*handle-list*`)`: wait until all asynchronous operations in the queues specified by *handle-list* have completed
- `wait async(`*handle*`)`: enters the synchronisation into the queue *handle*

```
[parallel | kernel | enter data | exit data |
]  [async[(handle)]] [wait[(handle-list)]]
```

- `async`: enters the operation into a default queue
- `async(`*handle*`)`: enters the operation into the queue *handle*
- `wait`: operation starts after all asynchronous operations have completed
- `wait(`*handle-list*`)`: operation starts after all asynchronous operations in the queues specified by *handle-list* have completed
- combinations possible, e.g. `parallel wait async(1)` enter parallel region into queue 1, but don't execute it until all asynchronous operations have completed

Asynchronous example 1

```
[prepare array a on CPU]
#pragma acc enter data async(1) copyin(a)
[prepare array b on CPU]
#pragma acc enter data async(2) copyin(b)
#pragma acc parallel loop async(1) present(a)
for(i=istart ;i<iend; i++) {
   a[i]= [some computation on GPU]
}
#pragma acc exit data copyout(a) async(1)
#pragma acc parallel loop async(2) present(b)
for(j=jstart ;j<jend; j++) {
   b[j]= [some computation on GPU]
}
#pragma acc exit data copyout(b) async(2)
[some computation on CPU]
#pragma acc wait
[continue to use updated a,b on CPU]
```

- simple example with two arrays
- update of arrays independent of each other
  - copy data to GPU
  - compute on GPU
  - copy back to CPU
- this approach can be generalized, e.g. for slices of a larger array

Asynchronous example 2

```
REAL::a(Nvec,Nchunks),b(Nvec,Nchunks)
!$acc data create(a,b)
DO j = 1,Nchunks
!$acc update device(a(:,j)) async(j)
!$acc parallel loop async(j)
   DO i = 1,Nvec
      b(i,j) = [function of a(i,j)]
   ENDDO
!$acc update host(b(:,j)) async(j)
ENDDO
!$acc wait
!$acc end data
```

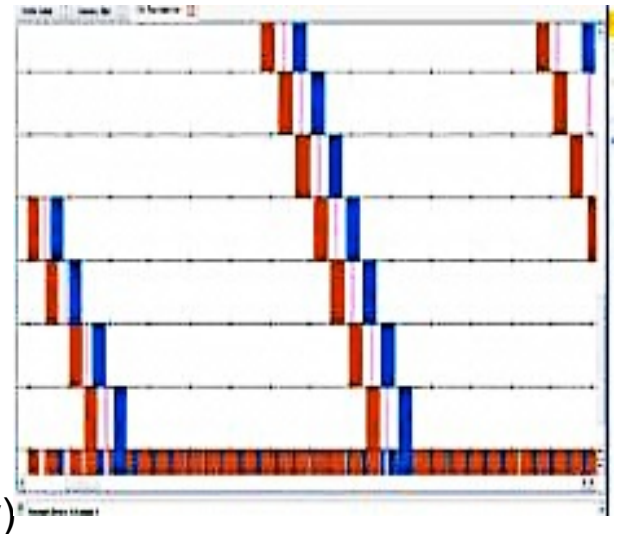
- can overlap 3 streams at once
- use slice number as stream handle
- don't worry if number gets too large
- OpenACC runtime maps it back into allowable range (using MOD function)

Execution times (on Cray XK6):

- CPU: 3.76s
- OpenACC, blocking: 1.10s
- OpenACC, async: 0.34s

NVIDIA Visual profiler:

- time flows left to right
- streams stacked vertically
- only 7 of 16 streams fit in window
- red: data transfer to GPU
- pink: computational on GPU
- blue: data transfer from GPU
- vertical slice shows what is overlapping
- collapsed view at bottom
- async handle modded by number of streams
- so see multiple coloured bars per stream (horizontally)

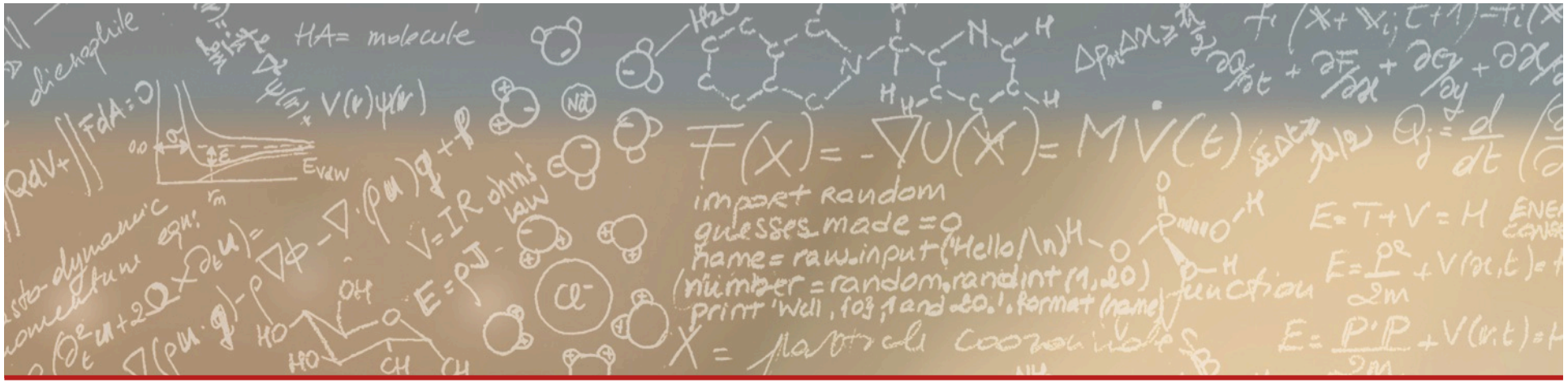


Recommendations for use of async

- view it as part of performance tuning
- first implement synchronous code, verify it
- investigate bottlenecks:
  - do the kernels need tuning?
  - do the data transfers need tuning?
- look for data independencies
  - across kernels
  - between kernels and host code
- once you have the extent of independent regions, add asynchronous clauses / directives
- careful with `async` handles: only integers and easy to confuse if you need many different ones
  - consider using e.g. named integer constants if reasonably descriptive naming is possible, e.g. to separate different sets of queues from one another

Some useful tips at the end…

- if in doubt, check the OpenACC standard
- focus on getting correct code on the GPU first
- then start optimizing
- focus on data transfers before aiming for a few percent improvement on a kernel
- on Cray systems, get detailed info about size of data transfers, kernels launched, etc:
  environment variable `CRAY_ACC_DEBUG=2`
- make efficient use of tools provided at your computing centre (e.g. DDT/totalview for debugging)
  - it might take some time to ‘learn’ using the tool
  - but debugging complex code with printf will cost you much more time
  - same goes for performance analysis !



Thank you for your attention.

Questions…?