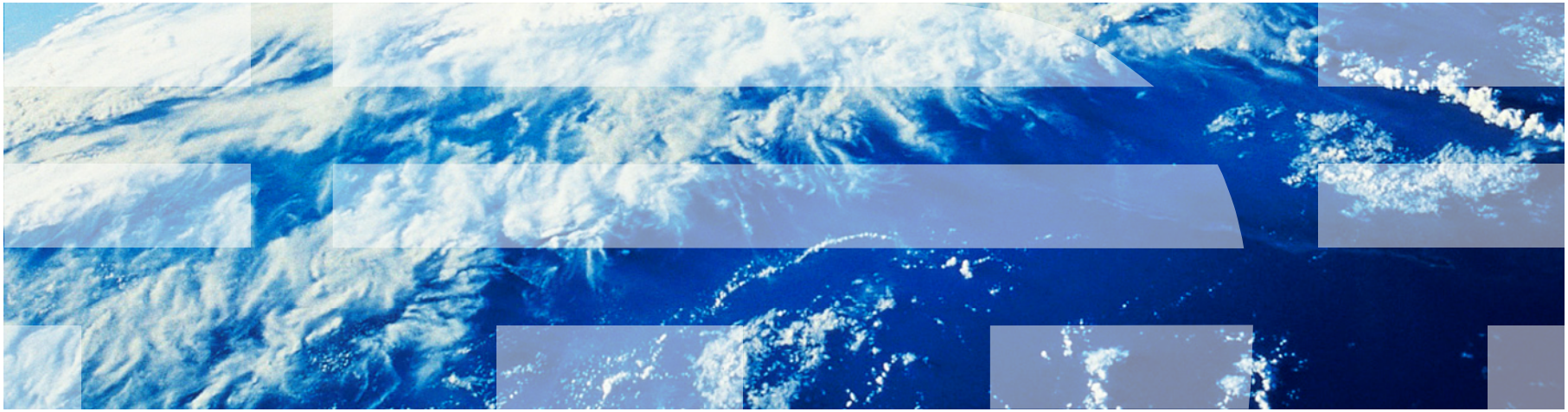
Alexandre Eichenberger

01/10/17

OpenMP 4.5

– Relevant Accelerator Features –



T.J. Watson, IBM Research

A Paradigm Change with OpenMP 4.5

- **Extracting maximum performance:**
  - to program a GPU: you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC
  - to program a host SIMD unit: you have to use Intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints)
  - to program the CPU threads, you might use C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors

- **With OpenMP 4.5 and up:**
  - you can use the same standard to program the GPU, the SIMD units, and the CPU threads

[sources: Michael Wong]

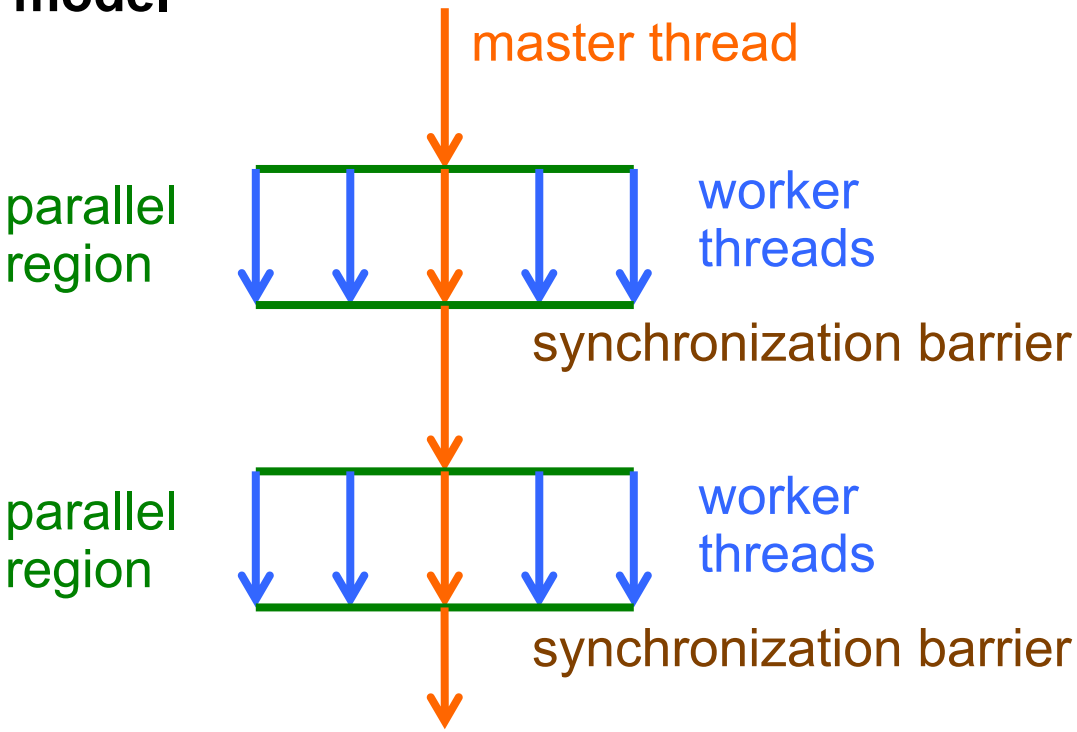
OpenMP 4+ Features

- **Directives**
  - parallel regions
    - thread affinity
  - worksharing
    - loop, sections,…
    - *ordered(do across)*
  - SIMD
  - tasking
    - loops, groups, dep, prio
  - accelerator (target)
    - unstructured, *nowait*
  - synchronization
  - cancellation
  - data attributes
    - shared, private [first/last]
    - [user] reductions
    - target: map data to/from
    - target: [first] private, subset
- **Environment Vars**
  - number of threads
  - scheduling type
  - dynamic thread adjustment
  - nested parallelism
  - thread limit
  - description of hardware thread affinity
  - thread affinity policy
  - default accelerator devices
- **Runtime Variables**
  - number of threads
  - thread id
  - dynamic thread adjustment
  - nested parallelism
  - schedule
  - active levels
  - thread limit
  - nesting level
  - team size
  - locks [*hint*]
  - mapping API

[*italic means in progress*]

OpenMP Execution Model for Parallel Regions

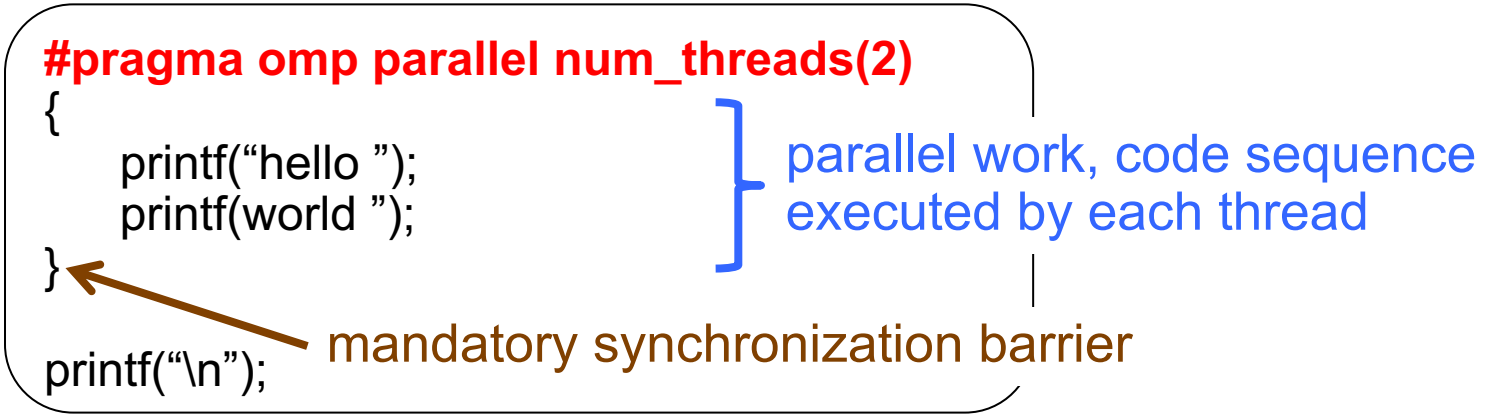
- **Fork and join model**



- **Model:**
  - sequential code executed by the master thread
  - parallel code executed by the master and workers
  - parallel region terminated by a synchronization barrier
  - memory touched in parallel region is “released/flushed” at barrier

Parallel Construct Example

- **Example**

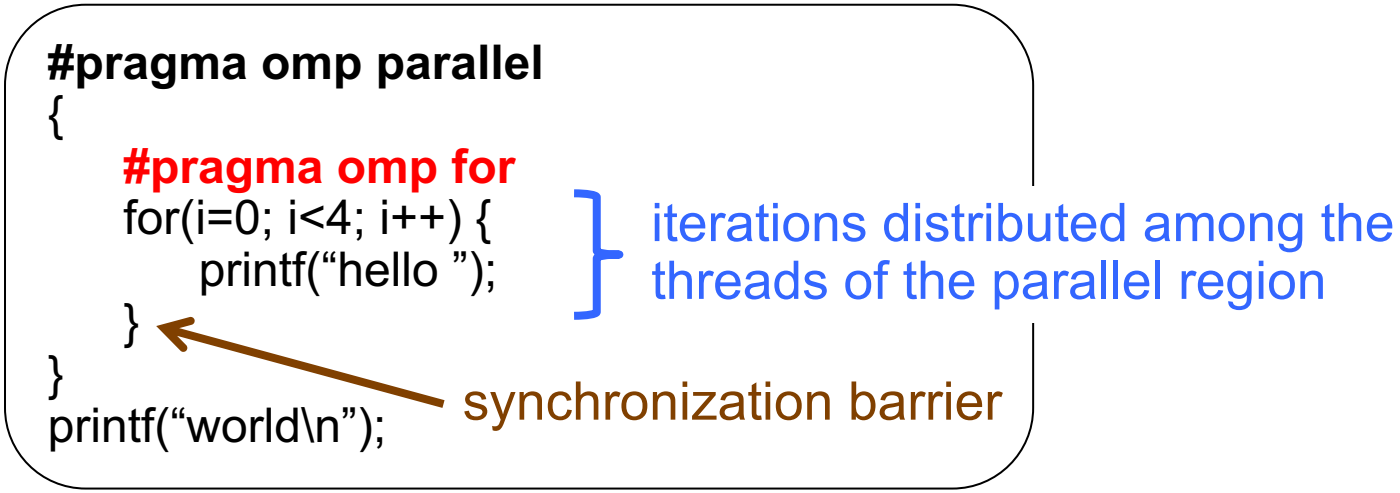


  - output: "hello world hello world" or "hello hello world world"

- **Additional clauses can modify the parallel region**
  - dealing with threads (if, num_threads, proc_bind)
  - dealing with data (shared, private, first-private)
  - dealing with reductions

Worksharing Construct for Loops

- **Worksharing constructs distribute work to threads of a parallel region**
  - examples are loops and sections



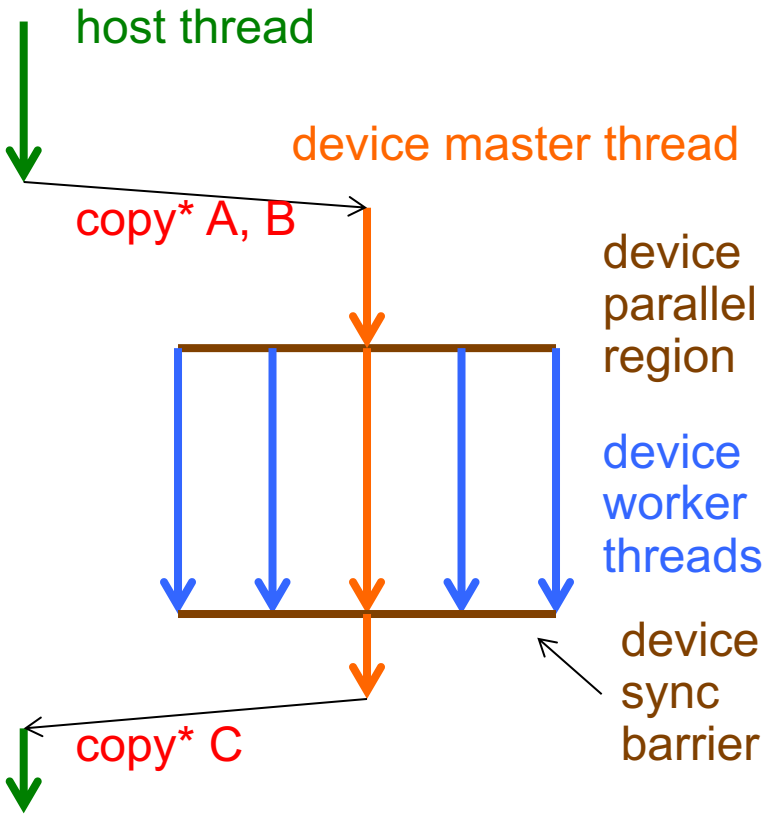
  - output: "hello hello hello hello world"

  - many of parallel region clauses apply here as well
    - e.g. shared, private, first-private, reduction
  - a few clauses are new
    - e.g. schedule, nowait, last-private, ordered

OpenMP Accelerator Overview

- **Simply add a target construct**

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target
    map(to: A, B) map(from: C)
{
     int n = 64;
     #pragma omp parallel for
     for(int i=0; i<n; i++) {
          for(int j=0; j<n; j++)
               for(k=0; k<n; k++)
                    C[i, j] = A[I, k] * B[k, j];
     }
}
```



– target transfer control of execution to a SINGLE device thread
– clause "**map**" are used to fine tune copying of data; default is "**map(tofrom:)**"

* at most one copy of each data structure exists on a device; outermost target map copies data to/from device, copies optional with unified memory

Difference with Typical GPU Programming Models

- **Traditional models: "Execute this one, exclusively-parallel loop"**
  - such as found in CUDA, OpenCL,…
  - transfer control to a single "parallel loop"
  - no sequential code (e.g. to initialize data serially on GPU)

- **OpenMP model: "Just another normal OpenMP program, on device"**
  - leverages every+ OpenMP construct
  - includes parallel regions, parallel loops, tasks, …
  - includes fine grain and coarse grain synchronizations within one team
    - e.g. locks, critical regions, barriers…
  - can have sequential and parallel code

- **OpenMP supports traditional model too:**
  - it is a "target teams distribute parallel for simd" combined construct

+ exception: target constructs cannot be nested

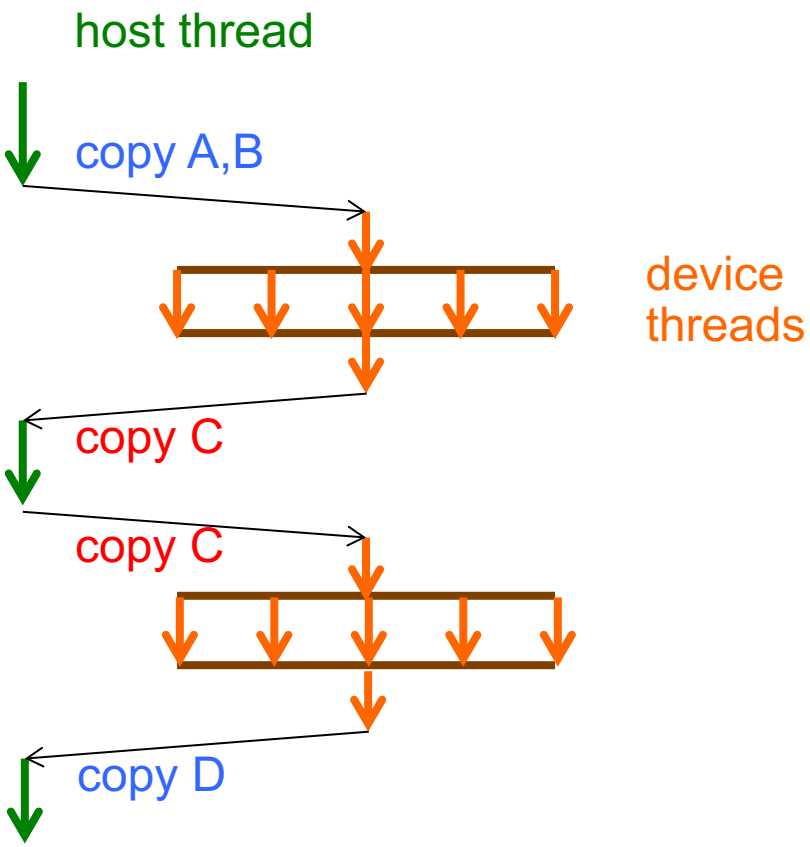
Target Data

- **Data scope & data movement**
  - approaches to minimizing transfers
- **Data types that can be mapped**
  - scalars, arrays, structs & classes
- **Memory Model**
  - handling unified vs. distributed memory

Overcoming Data Movement

- **Scope of data is important**

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target
    map(to: A, B) map(from: C)
{
     // define C in terms of A, B
}


#pragma omp target
    map(to: C) map(from: D)
{
     // define D in terms of C
}
```

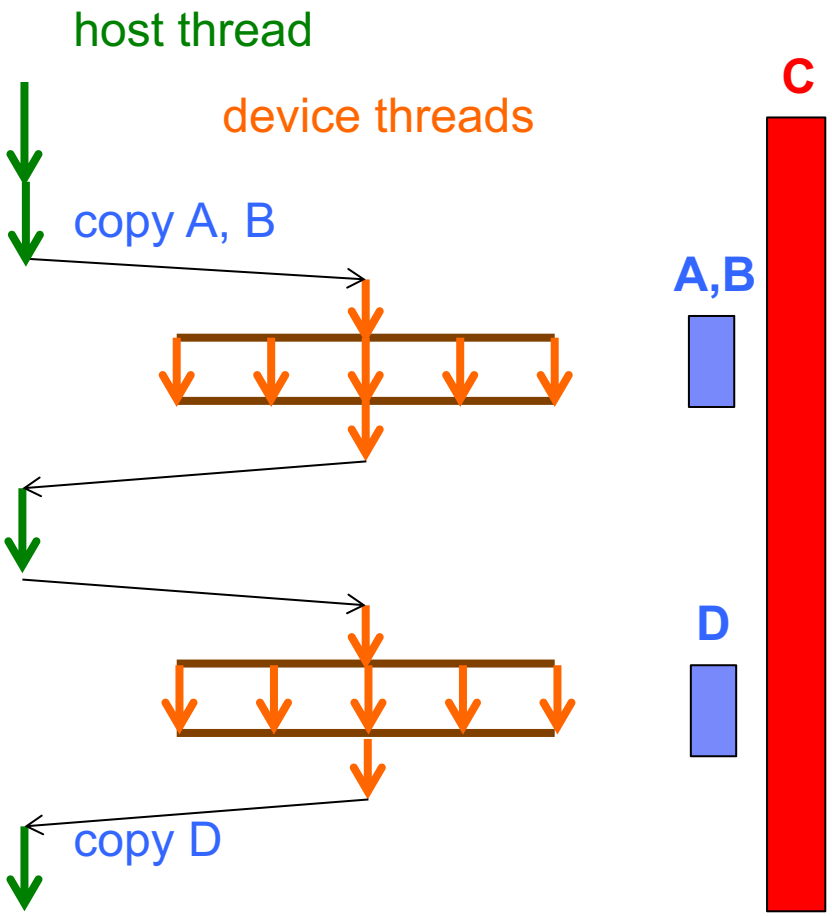


- data scope is limited by the target constructs
- no data scope for variable C between the two constructs on the device
- results in needless copies of C

Overcoming Data Movement (cont.)

- **Scope of data is important**

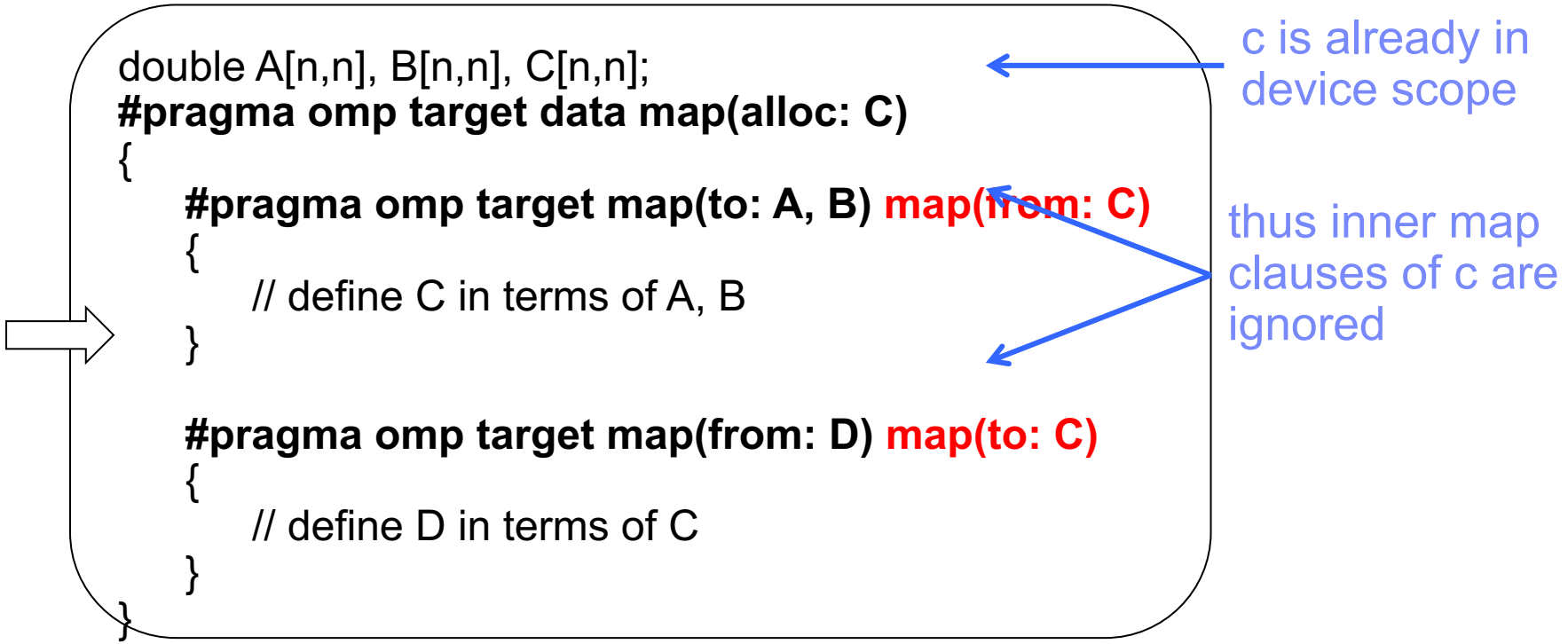
```
double A[n,n], B[n,n], C[n,n];
#pragma omp target data
    map(alloc: C)
{
    #pragma omp target
        map(to: A, B)
    {
        // define C in terms of A, B
    }

    #pragma omp target
        map(from: D)
    {
        // define D in terms of C
    }
}
```



  - C is now a temporary variable that remains on the device
  - C is not initialized on the device

Forcing Data Movement

- **Device has at most a single copy of each mapped variable**
  - map clauses are *ignored* when data is already in device scope



  - add "**#pragma omp target update from(C)**" force a copy back to the host
  - or use "**always**" qualifier in the map clause, e.g. "**map(always from: C)**"

Data Always Residing on Accelerator

- **Static data**
  - use “target declare” to create a resident copy
  - if need to move back and forth, can use “target update”

```
#pragma omp declare target
double A[100];
int *p;
#pragma omp end declare target

#pragma omp target
{
     A[20] = 100;
     p = malloc(10*sizeof(int));
}
#pragma omp target update from(A)
```

- **Dynamic data**
  - use “target declare” for pointer to data structure
  - use malloc within target regions to populate the pointer
  - cannot bring pack the dynamic data (not mapped)

Summary of Data Scope

- **Scope linked with device execution: target**
  - **"#pragma omp target map(x) {…}"**
  - defines a data scope for the duration of execution on device

- **Pure Scope, without associated device execution: target data**
  - **"#pragma omp target data map(x) {…}"**
  - only defines a data scope, without launching execution on device

- **User can also declare data on the device**
  - **"#pragma omp declare target to(x)"**
  - **"#pragma omp declare target" … "#pragma omp end declare target"**
  - user is responsible to move data back and forth (except for static initialization)

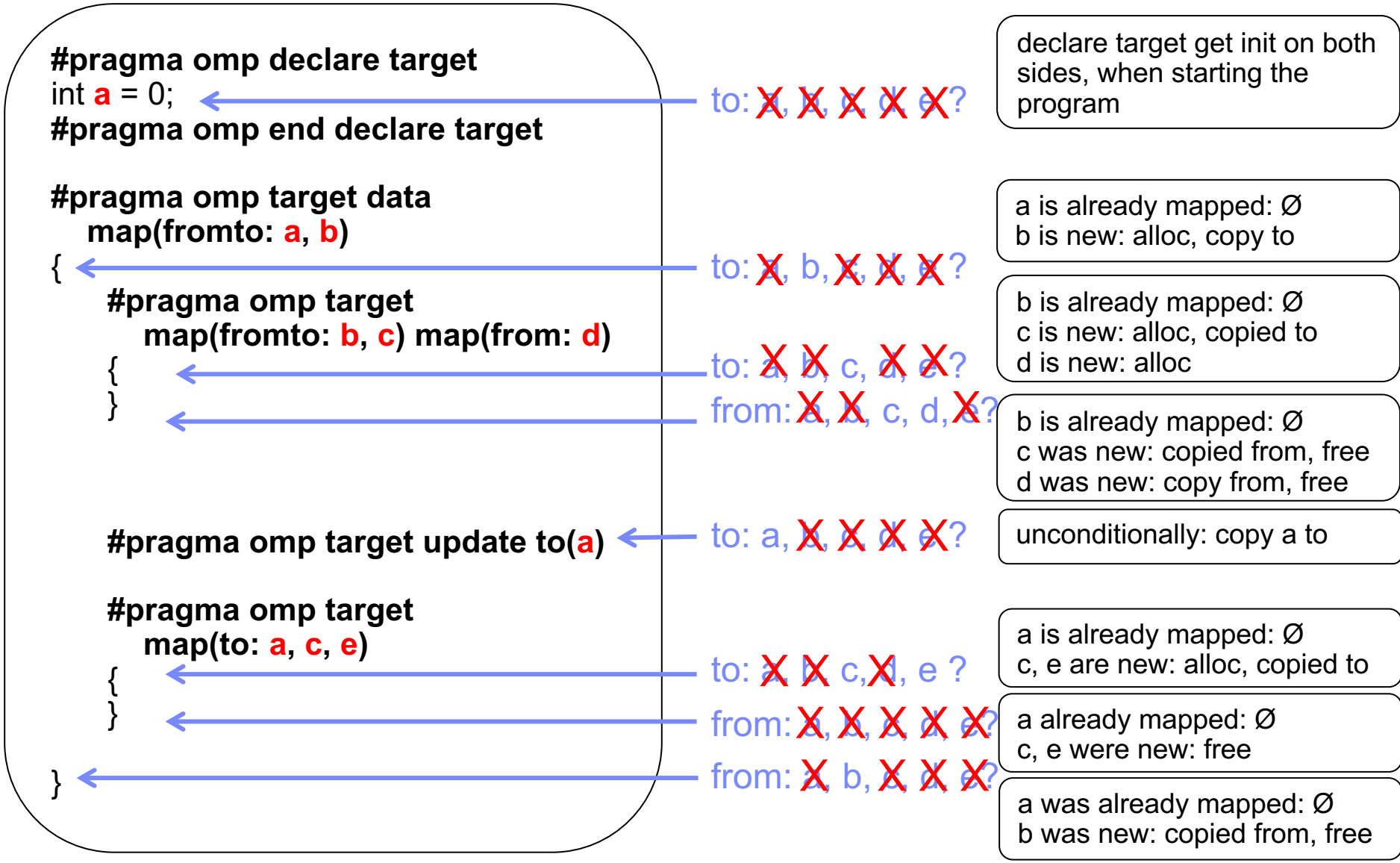
- **Unstructured pure scopes: target entry/exit**
  - "**pragma omp target enter/exit data map(x)**"
  - unstructured scope, can be inserted anywhere while executing on the host

In Details: When Are Values Consistent?

- **Values are consistent between the host and a device after:**
  - top level target* map clause with to / from
  - target* map clause with to / from & always qualifier
  - target exit data with a delete clause
  - target update with to/from clause
- **Top Level?**
  - we have a ref count for each map
  - incremented by 1 for each of the following constructs
    - on entry of a target / target data
    - target enter data
  - decremented by 1 for each of the following constructs
    - on exit of a target / target data
    - target enter data
  - exceptions?
    - maps associated with target declare have infinite ref counts
    - target exit data have a “delete” clause that forces ref counts to zero

* specifically: target, target data, target enter data, target exit data

Putting it Together: When Copy May Occur?



Other Data Clauses on Target

- **Private**
  - makes a private copy on the device
- **First private**
  - makes a private copy on the device, initialized with the host value
- **Interactions with mapped data**
  - a variable cannot be in a map and a private clause on the same construct

Target Data Mapping Types

Data Mapping Types: Basic Types

- **Mapping maintains one copy of data per device**
- **Scalar, whole arrays**
  - int a, B[100], C[1000];
  - **#pragma omp target map(a, B)**
- **Array shaping**
  - can shape an array to only load a subset of it with “[offset: length]” syntax
  - **#pragma omp target map(C[500:100])**
- **Restrictions:**
  - a name (e.g. “C” above) can only be used in one array shaping
    - e.g. the following is forbidden: map(C[0:100], C[500:100])
    - different names can be used (e.g. int *p = &C[0], *q = &C[500])
  - a shape can only be resized to a smaller shape
    - e.g. map(C[0:200]) is illegal if map(C[0:100]) is already mapped in context

Data Mapping Types: Pointers

- **Pointers should be qualified**
  - otherwise, the runtime doesn't know what amount of data to allocate and move
  - pointer is a firstprivate (modification of pointer on device are not reflected back)

```
- int *p = malloc(100*sizeof(int))
- #pragma omp target map(p[0:100])
```

- **Unqualified pointers will attempt to locate data**
  - if the pointer points to mapped data (at offset 0, for length of 0 bytes),
    - it will be set to that mapped data
  - otherwise, it will be set to NULL

```
int A[N], B[N], *p;
p = x ? &A : &B;
#pragma omp target data map(A)
#pragma omp target map(p[0:0])
{
    printf("0xllx\n", p);   // x ? map of A : NULL
}
```

Data Mapping Types: Struct

- **Structs can be mapped in their entirety**
  - entire struct is bitwise copied (i.e. pointer p contains an host pointer address)
  - struct {int a, B[100], *p, pNum;} s, S[10];
  - **pragma omp target map(s, S);**

- **Subset of structs can be mapped**
  - best way to understand how? Array subsection analogy.
  - think as is each struct element is a name for an array element.
  - **pragma omp target map(s.a, s.B);**
  - **pragma omp target map(s.pNum, s.p[0:s.pNum]);**

- **Restrictions**
  - can only access subfields that were explicitly mapped
    - if one subfield is explicitly mapped, they must all be explicitly mapped
    - if no subfield is explicitly mapped, the the whole struct is mapped
  - cannot add additional subfields in an enclosed target constructs

Data Mapping Types: Classes

- **Map an entire class to the device**
  - using target declare on the whole class using target declare target
  - class can then be used on host or device
  - constructor/destructor used on host/device, but bitwise copy

```
#pragma omp declare target
class A { public: int a, b; int foo() {return a+b;} };
#pragma omp end declare target
```

- **New for OpenMP 4.5 – TR4 (preliminary for 5.0**
  - can also map individual methods
  - can map class static variables
  - can map class that have virtual functions, as long as they are not used on the device

  - still has issue using the map(*this)… onging work to make it happen

Data Mapping Types: Classes (cont.)

- **Introduce target constructs in methods of a class**
  - within a method, can introduce a target construct
  - class instance variables are mapped as struct elements

```
class A { public: int a, b, sum;
  int foo() {
    #pragma omp target // implicit map(this->a, this->b, this->sum}
      sum = a + b;
    return sum;
}};
```

Default Mapping

- **References used in the target region have a default mapping**
  - references used outside of the target region (e.g. in a called function) must be explicitly mapped, or included in a declare target region.

- **Default for arrays: map(tofrom: A)**
  - array are mapped in their entirety

- **Default for scalars: firstprivate(i)**
  - default can be changed using “defaultmap(tofrom:scalar)” clause

- **Default for pointers: map(tofrom: p[0:0])**
  - default is zero-length pointers

Default Scalar Mapping Pitfalls

- **Pitfalls with scalars:**
  - data that you want back will not be seen with firstprivate

```
int A[N], sum=0, i;
#pragma omp target
  #pragma omp teams distribute parallel for reduction(+:sum)
     for(i=0; i<N; i++) sum += A[i];
```

  - should write this instead [temporary fix]

```
int A[N], sum=0;
#pragma omp target map(sum)
  #pragma omp teams distribute parallel for reduction(+:sum)
     for( int i=0; i<N; i++) sum += A[i];
```

  - new to TR4: reduction clause on map, which will negate the need to map sum

Default Pointer Mapping Pitfalls

- **Pitfalls with pointers:**
  - pointers will become implicit zero-length arrays
  - result in NULL value on the device if data is not mapped

```
int A[N], *p; p = &A;
#pragma omp target map(p[0:0])
   for(i=0; i<N; i++) p[i]++;
```

  - should write this instead

```
int A[N], *p; p = &A;
#pragma omp target data map(A) // at some earlier time
   #pragma omp target map(p[0:0])
      for(i=0; i<N; i++) p[i]++;
```

  - "export XLSMPOPTS=mapwarning=on" will emit warning for NULL pointers

Default Pointer Mapping Pitfalls (cont.)

- **These map pairs are not equivalent:**
  - map(p[:]) maps the pointer to data mapped before (or NULL if not)
  - map(p) maps the pointer as a scalar, copy host address on the device

  - map(p[:N]) maps an array of N elements
  - map(P[N]) maps the Nth element

Implicit Scalar-Map Pitfall

- **Mapping a scalar in outer-scope does not change implicit rules**

```
int sum = 0
#pragma omp target data map(sum)
{
    #pragma omp target   // implicit map of sum -> first private
     sum++;
}
```

  - while it is true that sum is mapped in the target data
  - it is implicit on the target
    - target does not “care” that it was previously mapped
    - it will be first private

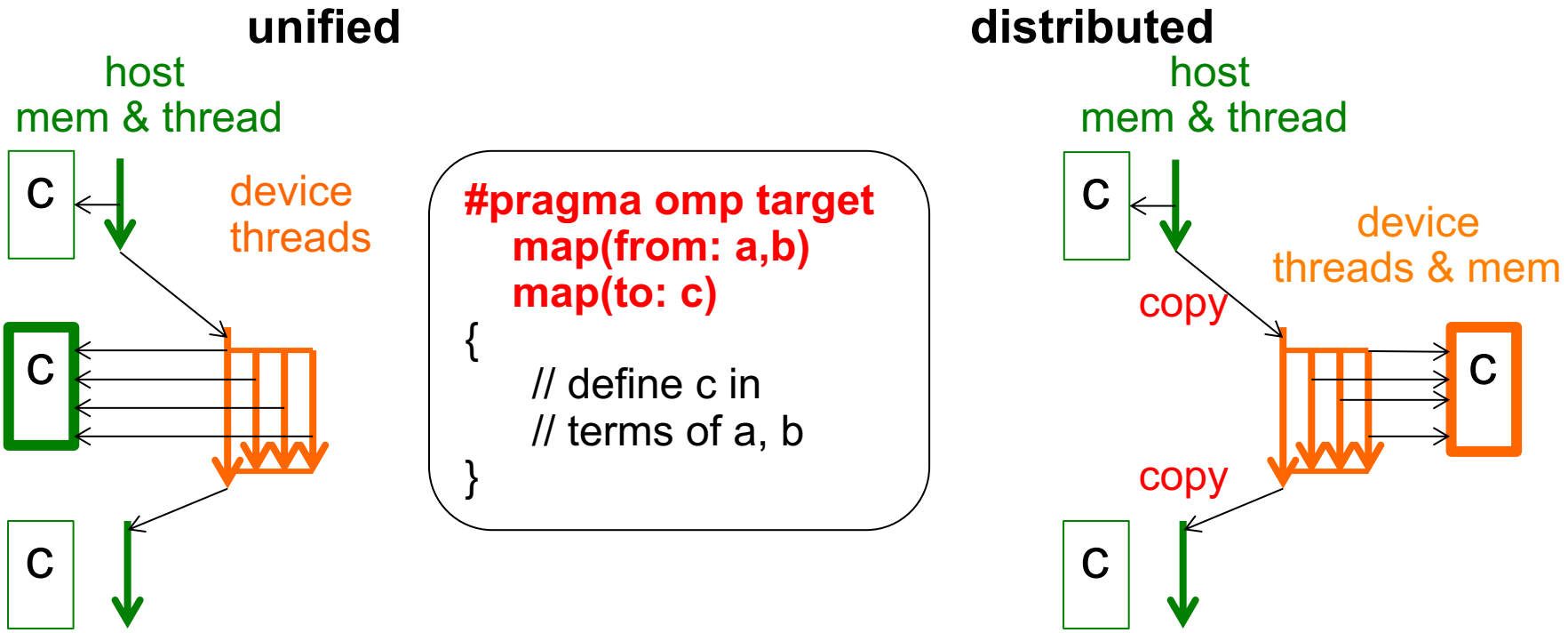
- **Correct**

```
int sum = 0
#pragma omp target data map(sum)
{
    #pragma omp target map(sum)
     sum++;
}
```

Target Memory Model

Accelerator Memory Model

- **Programmers may not assume which model is used**



  - so the values of c may (unified) or may not (distributed) change during target execution
  - user should not assume one or the other in a valid OpenMP program

Accelerator Memory Model: Valid Program

- **Different results depending on memory model: not a valid program**

- **How to write a legal OpenMP program:**
  - must schedule a ‘target update’ or ’target map(always:)’
    - each time that a value def/used on one device
    - and then def/used on another device
  - [use/use pattern is fine without intervening target update/map always]

Accelerators with Unified Memory

- **Map clause does not need to copy data to device private memory**
  - since it can access shared memory
  - user must still have them…

- **But we may decide to selectively copy data**
  - e.g. read only data accessed by both host and accelerators
    - without copy: may generate misses if not cacheable in both
    - with explicit copy: no misses
  - e.g. dense array may be copied over
    - single DMA moves all of the data
  - e.g. data structures with pointers may not be copied over
    - to “deep copy” (feature not avail as of now) a linked list, one needs to DMA each element of the list to the device, update all of the pointers, … and they may not be used anyway

Target Execution Model

Target & Target Teams

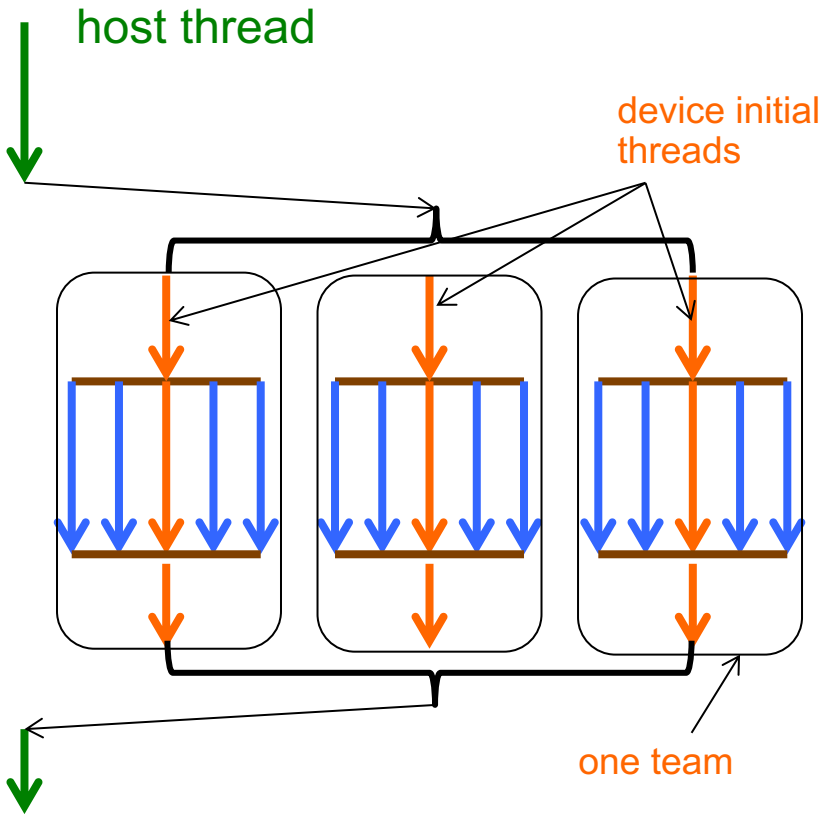
- **Target constructs**
  - start a single team of threads
  - single initial thread executes until encountering a parallel construct
  - **#pragma omp target**

- **Target teams constructs**
  - start a league of teams (of threads)
  - teams cannot synchronize, but we can have reductions over all teams
    - atomic operations on data no larger than 64 bits allowed
  - one initial thread executes, per team (just like threads in a parallel)
  - execution can diverge when executing a “distribute” (just like “for” in a parallel)
  - standalone construct
    - must be directly nested within a target construct
    - **#pragma omp teams**, directly nested inside
  - combined constructs, e.g.
    - **#pragma omp target teams**

Target Teams

- **Combined target teams construct**

```
#pragma omp target teams
    map(to: a, b) map(from: c)
{
     int n = 64;
     #pragma omp distribute
     for(int i=0; i<n; i++) {
          #pragma omp parallel for
          for(int j=0; j<n; j++)
               for(k=0; k<n; k++)
                    c[i, j] = a[I, k] * b[k,
j];
     }
}
```



- target transfer control of execution to one device thread per team
- every team initially execute the same code
- in a “#pragma omp distribute”, each team get it’s subset of iteration space

Teams Clauses

- **Several clauses can be used with teams (not target)**
  - num_teams(int): number of desired teams in the league
  - thread_limit(int): upper bound on the number of threads per team
  - reduction(reduction identifier: var list):
    - reduction to be performed by all teams in the league
    - reduced value available on the host after the target construct is complete

Distribute Construct

- **Distribute work of a loop among teams**
  - similar to “#pragma omp for”, but for teams instead of threads
  - fewer schedule policies (only static at this time)
  - has same data attributes (private, firstprivate, lastprivate) and collapse
  - but no barrier or reductions at the end of the distribute

Reductions

- **Reductions can be added on**
  - teams
  - distribute
  - parallel
  - for
  - simd

Optimized Constructs

- **Current construct that are optimized (special pattern)**
  - target teams distribute parallel for
  - teams distribute parallel for

  - result in a much optimized pattern, for compiler and runtime.

Interaction with Device Native Routines

- **On target data, use_device_ptr(p)**
  - allows the user to extract the device address of a mapped variable
  - can be used to call native routines of a device (e.g. Cuda kernels)

- **On target, is_device_ptr(p)**
  - allows the user to give a device computation the address of a device buffer

- **Typical use**

```
int A[N];
#pragma omp target data map(A) use_device_ptr(A)
 {
     cuda_step1(A);
     #pragma omp target is_device_ptr(A)
     { /* step 2 */ }
     cuda_step3(A);
}
```

Runtime Routines & Environment Variables

Compiler Flags

-fopenmp-nonaliased-maps

- flag enables non-coherent texture loads

Environment Variables

- **OMP_DEFAULT_DEVICE**
  - set default device, when "**device(num)**" clause is not specified
- **OMP_TEAMS_LIMIT [non-standard]**
  - set maximum number of teams, override "**num_teams(num)**" when too large
- **OMP_NUM_TEAMS [non-standard]**
  - set default number of teams, when "**num_teams(num)**" is not specified
- **XLSMPOPTS=' TARGETTHREADLIMIT=num'**
  - set the maximum number of threads on a target, if thread_limit is not specified
- **XLSMPOPTS='TARGETNUMTHREADS=num'**
  - set default number of threads on a target
- **XLSMPOPTS=' MAPWARNING=ON'**
  - emit runtime warning when a zero-length pointer is not mapped
- **XLSMPOPTS=' TARGET=MANDATORY | DISABLED | OPTIONAL'**
  - force running on the device, disable running on the device, or run if possible

Environment Variables

- **Other host environment variables have no impact on devices**

- **When target code works on host:**
  - need OMP_NESTED to parallelize teams and parallel on host
  - need OMP_NUM_THREADS to specifies number of host threads
  - need OMP_PROC_BIND to bind threads to hardware
  - need OMP_PLACES to specify machine configuration (esp. with MPI)

Runtime Routine

- **Set/get default device (host only)**
  - void omp_set_default_device(int num);
  - int omp_get_default_device();
- **Get number of target devices (host only)**
  - int omp_get_num_devices();
- **Are we executing on the host? (host/targets)**
  - int omp_is_initial_device();
- **Getting team info (host/targets)**
  - int omp_get_num_teams();
  - int omp_get_team_num();

Device Memory Routine

- **Manually handle maps via routines**
- **Alloc/Free memory**
  - void* omp_target_alloc(size_t size, int device_num);
  - void omp_target_free(void * device_ptr, int device_num);
- **Maps**
  - int omp_target_is_present(void * ptr, int device_num);
  - int omp_target_associate_ptr(void * host_ptr, void * device_ptr, size_t size, size_t device_offset, int device_num);
  - int omp_target_disassociate_ptr(void * ptr, int device_num);
- **Memory move**
  - int omp_target_memcpy(void * dst, void * src, size_t length, size_t dst_offset, size_t src_offset, int dst_device_num, int src_device_num);
- **Host device number [to be used in device_num here]**
  - int omp_get_initial_device()

Concluding Remarks

- **OpenMP 4+ has lots of support for accelerators**

- **Our implementation is 4.5 compliant**
  - still working on a few bugs & issues

- **Our implementation is progressing toward high performance**
  - initial focus is on functionality
  - have started, and will continue, to tune for higher performance
  - have started, and will continue, to work on performance portability

- **Objectives of this meeting, from the compiler's perspective**
  - feedback on language features, wish list,…
  - getting micro-benchmarks for performance tuning