Argonne NATIONAL LABORATORY

OPENMP 4.5 DEVICE OFFLOADING DETAILS

COLLEEN BERTONI

Main references:

- **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017
- 4.5/5.0 OpenMP Specification and Examples

1. What is the basic device execution model, and the constructs used
   - Host-centric, hierarchical parallelism with thread teams
   - Difference between teams and parallel
2. How to distribute work to threads using worksharing, and the constructs used
   - Distribute, for/do, combined constructs
3. How to map data between host and device, constructs used, and data scopes for the main constructs
   - How to decrease unnecessary data transfer
4. How to check if what you think is happening is actually happening
   - OpenMP runtime routines
   - Nvprof (on Nvidia GPUs)

OVERVIEW

- Introduction and some terminology
  - Execution model and data environment
- Important OpenMP 4.5 Constructs/Concepts
  1. Device execution control
  2. Workshare
  3. Data mapping
  4. Runtime routines (and nvprof)
- Demo on JLSE at ALCF

INTRO AND QUICK EXAMPLE

Distributes iterations to the threads, where each thread uses SIMD parallelism

```
void vec_mult(float*p, float*v1, float*v2, int N)
{
  int i;
  init(v1, v2, N);
  #pragma omp target teams distribute parallel for simd \
                          map(to: v1[0:N], v2[0:N]) map(from: p[0:N])
  for (i=0; i<N; i++)
  {
      p[i] = v1[i]*v2[i];
  }
  output(p, N);
}
```

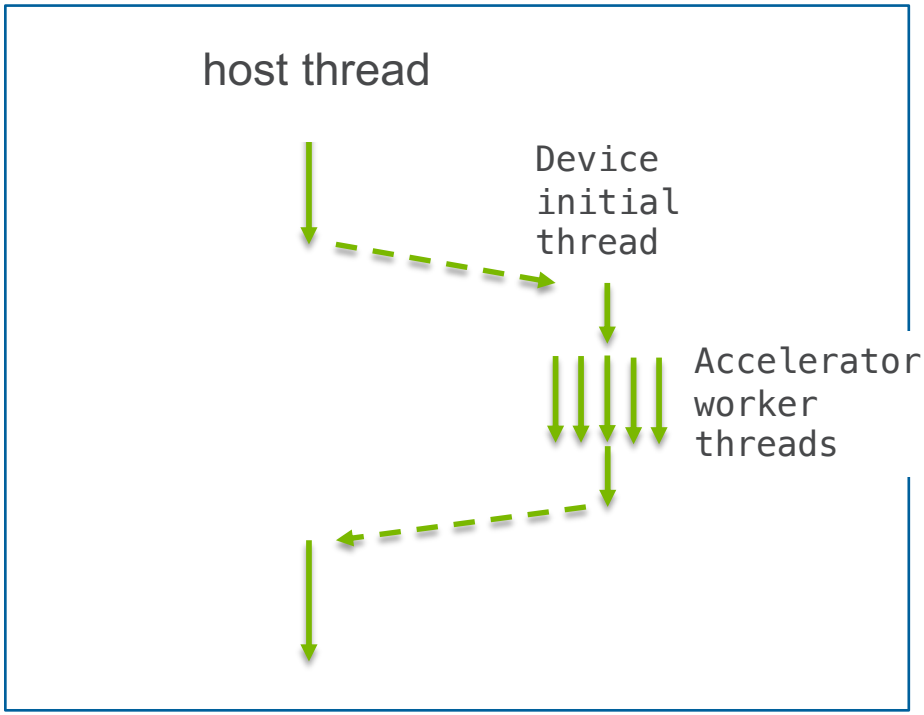
Creates teams of threads in the target device

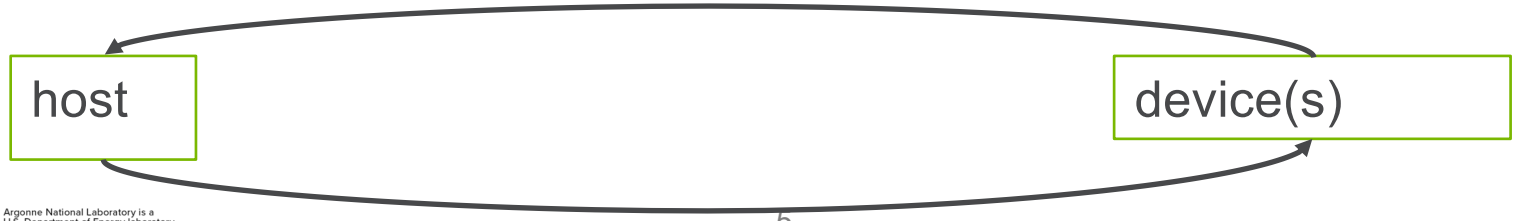
Controlling data transfer

Directive-based approach, different from OpenCL/CUDA

OVERVIEW OF EXECUTION

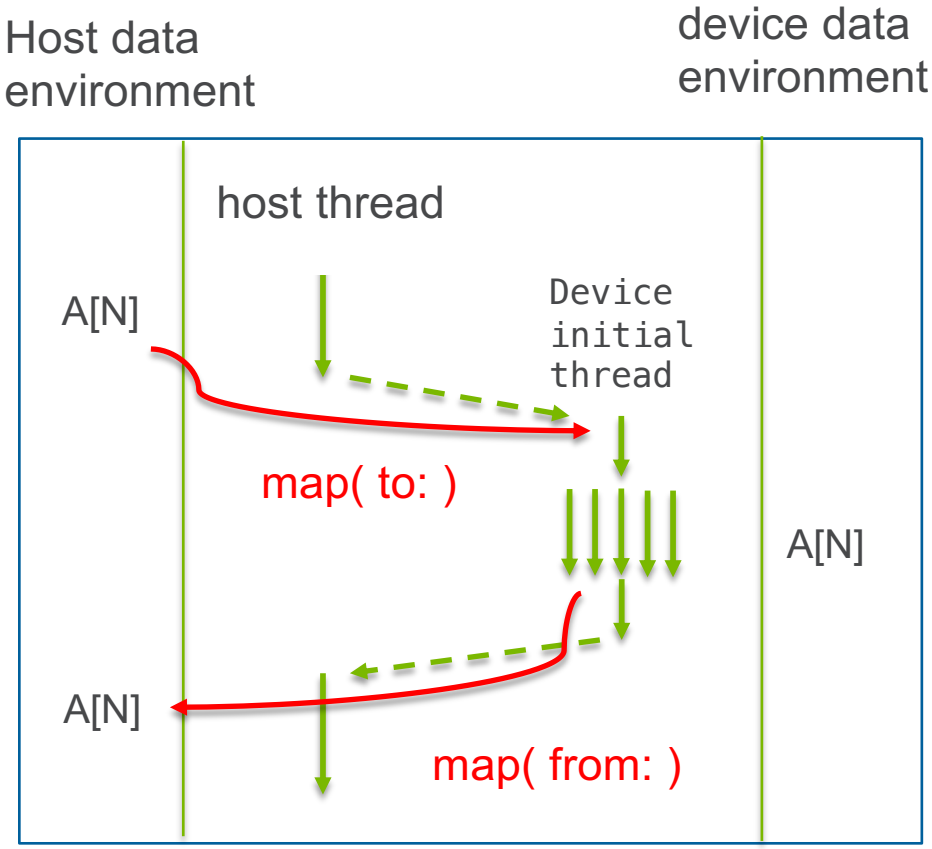
- Host-centric execution model
- *Device*: implementation-defined execution unit
- When the host thread reaches a target region, it generates a target task, and the host thread suspends the generating task it was executing
- Initial task is generated on the device
- Initial thread carries out the task on the accelerator
- When it's done, the host thread resumes





DATA ENVIRONMENT

- Accelerator contains a *device data environment,* which contains the set of all variables that are available to the threads on the accelerator
- When an *original variable* in the host's data environment is mapped to an accelerator, a *corresponding variable* is allocated in the accelerator's device data environment. Over the execution of the program, variables changes through mapping and unmapping
- Original and corresponding variables may or may not share the same storage location



OVERVIEW

- Introduction and some terminology
  - Execution model and data environment
- Important OpenMP 4.5 Constructs/Concepts
  1. Device execution control
  2. Workshare
  3. Data mapping
  4. Runtime routines (and nvprof)
- Demo on JLSE at ALCF

1. DEVICE EXECUTION CONTROL

DEVICE EXECUTION CONTROL

- During execution, we want to offload code to the accelerator, spawn threads to run code blocks in parallel, and take advantage of the available hardware
- Hierarchical parallelism
  - Fine-grained parallelism between threads in a single threadblock on a streaming multiprocessor (SM), share local memory and ability to sync
  - Coarse-grained parallelism different threadblocks running on the same SM or different SMs, share global memory

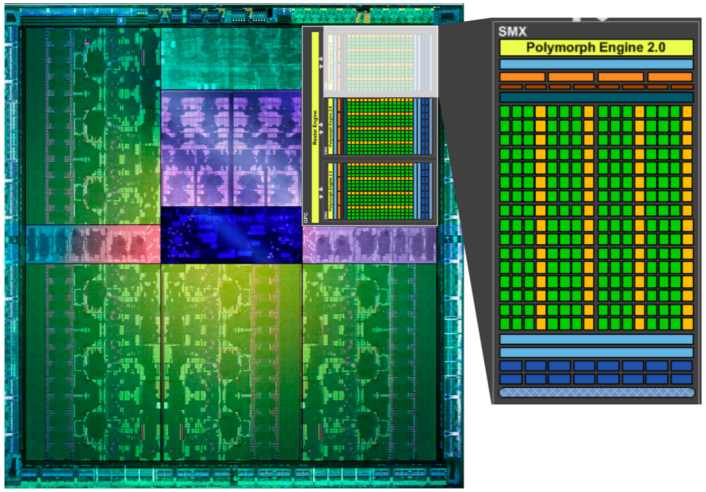


Image from Ref. 5 (“Targeting GPUs with OpenMP4.5 Device Directives”, Beyer and Larkin)

DEVICE EXECUTION CONTROL: IMPORTANT CONSTRUCTS

- During execution, we want to offload code to the accelerator, spawn threads to run code blocks in parallel, and take advantage of the available hardware

- Target construct

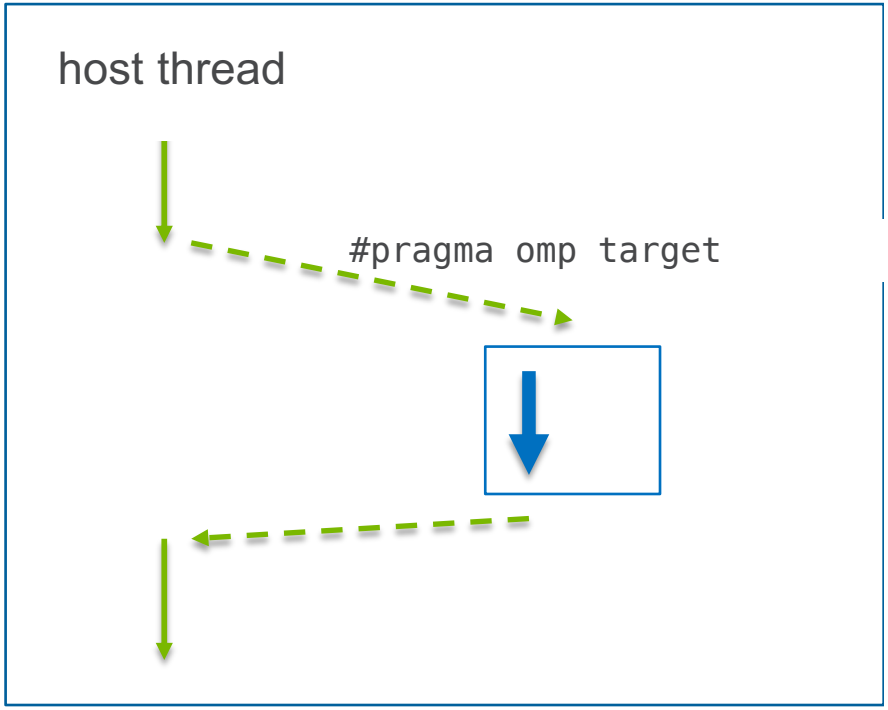
```
#pragma omp target [clause [[,]] clause ... ]
    structured block
```

- Target teams construct

```
#pragma omp teams [clause [[,]] clause ... ]
    structured block
```

TARGET CONSTRUCT

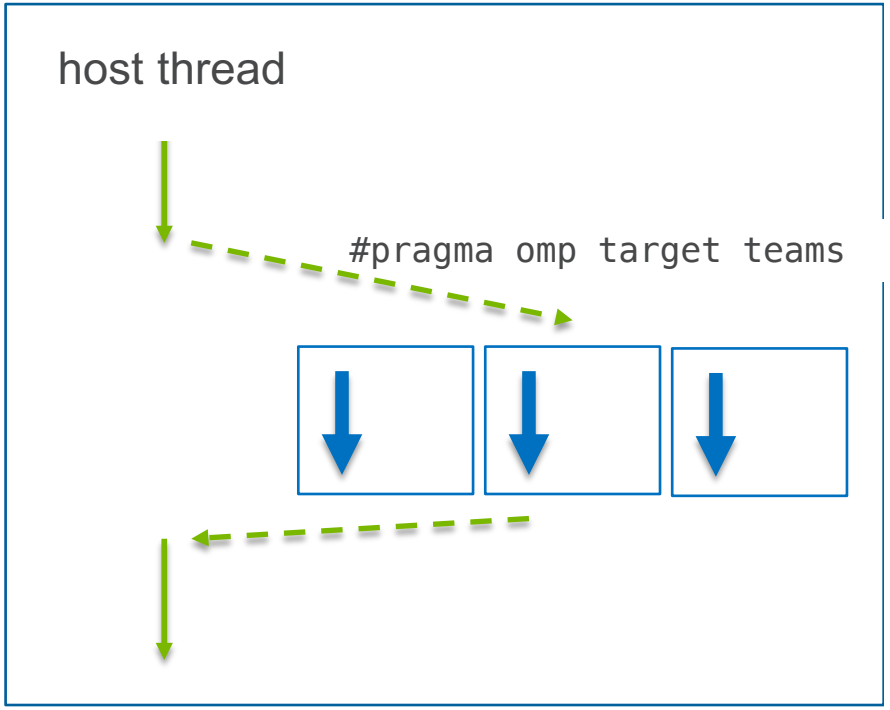
- Marks code for offload onto a device
- When a host thread reaches a target construct, the host thread execution pauses (by default) and a single initial thread executes the target region on the default device
- Clauses to control behavior, like nowait and device




```
#pragma omp target
{
    C = A + B;
}
```

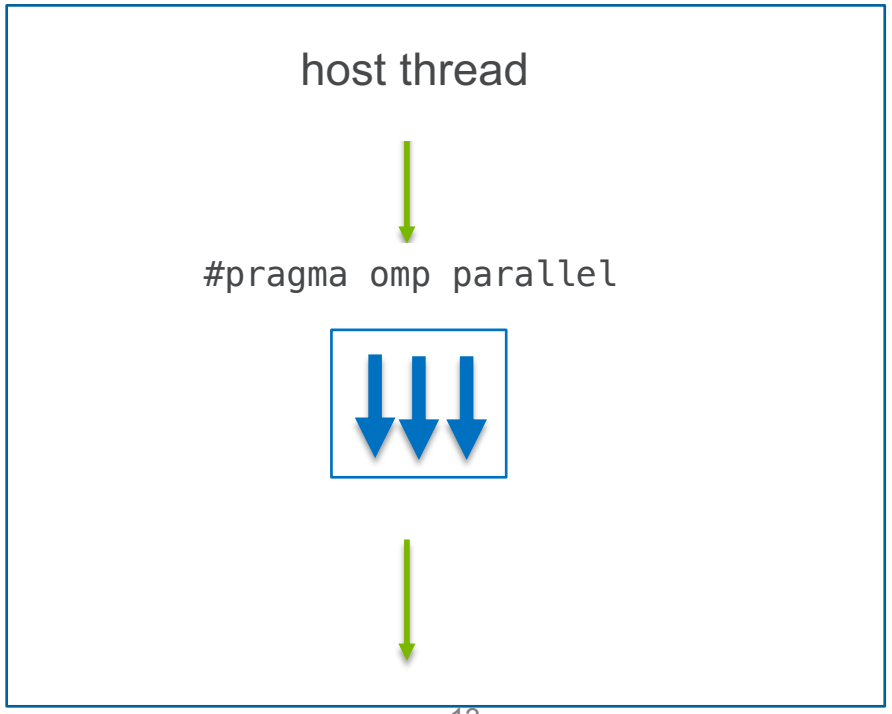
TARGET TEAMS CONSTRUCT

- **`target teams`** starts a *league* of initial threads where each thread is its own team, and in its own *contention group*. Each initial thread executes the teams region in parallel.
- Threads in different contention groups *cannot synchronize* with each other
- Different from the **`parallel`** construct, which creates a single team of threads, where each thread in the team executes the parallel region. Threads can synchronize with each other.
- Clauses to control behavior, like num_teams, thread_limit



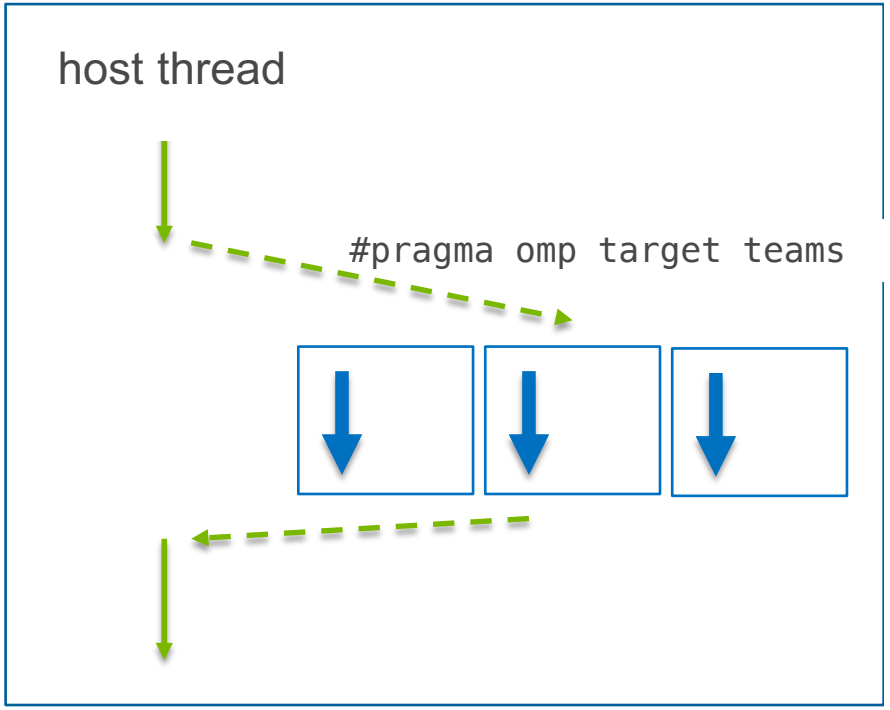
QUICK REVIEW: #PRAGMA OMP PARALLEL

- **`parallel`** construct, which creates a single team of threads, where each thread in the team executes the parallel region. Threads can synchronize with each other.



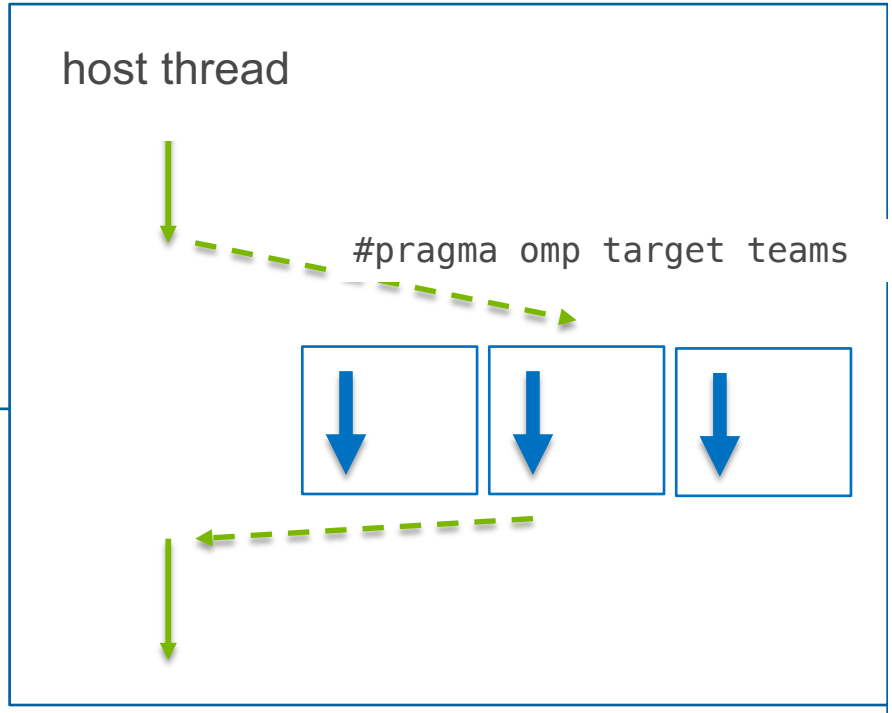
TARGET TEAMS CONSTRUCT

- **`target teams`** starts a *league* of initial threads where each thread is its own team, and in its own *contention group*. Each initial thread executes the teams region in parallel.
- Threads in different contention groups *cannot synchronize* with each other
- Different from the **`parallel`** construct, which creates a single team of threads, where each thread in the team executes the parallel region. Threads can synchronize with each other.
- Clauses to control behavior, like num_teams, thread_limit



TARGET TEAMS CONSTRUCT

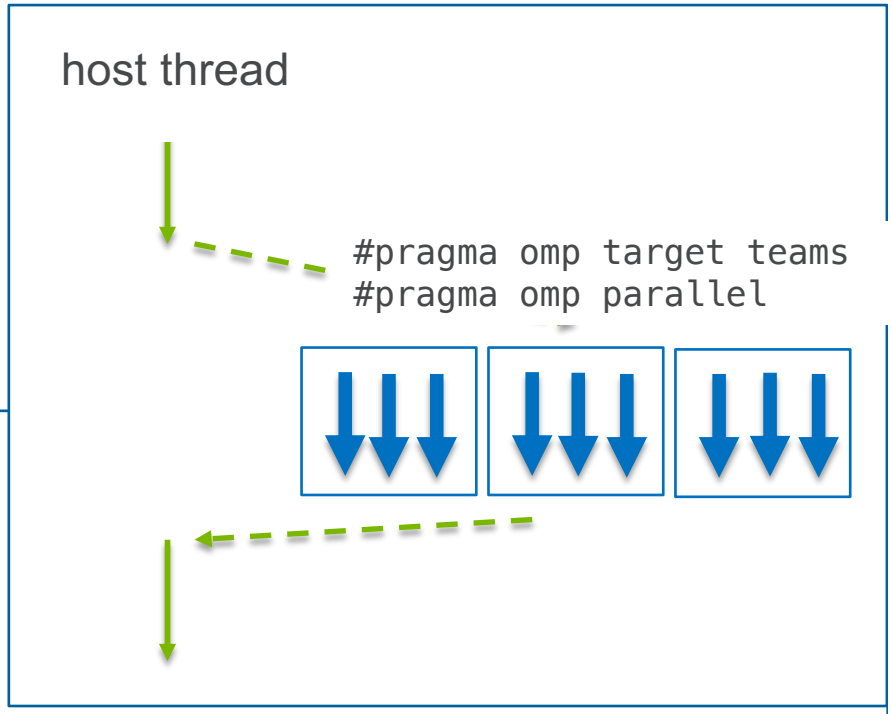
- The **target teams** construct creates a league of initial threads, where each thread is its own team
- Each team has only one thread




```
#pragma omp target teams num_teams(3)
{

    int team = omp_get_team_num();
    int nteams = omp_get_num_teams();

    printf( ”Hello from team %d out of %d teams.\n", team, nteams );

}
```

TARGET TEAMS CONSTRUCT

When a **`parallel`** construct is reached by a league, each initial thread becomes the master of a new team of threads, and each team concurrently executes the parallel region




```
#pragma omp target teams num_teams(3)
#pragma omp parallel
{
    int team = omp_get_team_num();
    int nteams = omp_get_num_teams();
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    printf( "Team %d out of %d teams\nThread %d of out %d threads in the team.\n",
            team, nteams, tid, nthreads );
}
```

SUMMARY: DIFFERENCE BETWEEN TEAMS AND PARALLEL

- #pragma omp teams
  - Coarser-grained parallelism
  - Spawns multiple teams, each with one thread
  - (Typically) map to SMs in Nvidia HW
  - Threads in different teams can't synchronize with each other
- #pragma omp parallel
  - Finer-grained parallelism
  - Spawns many threads in a team
  - (Typically) map to CUDA cores in Nvidia HW
  - Threads in a team can synchronize with each other (#pragma omp barrier)

2. WORKSHARING

WORKSHARING: IMPORTANT CONSTRUCTS

- Purpose is spread the iterations of a loop across available hardware resources
- Distribute construct

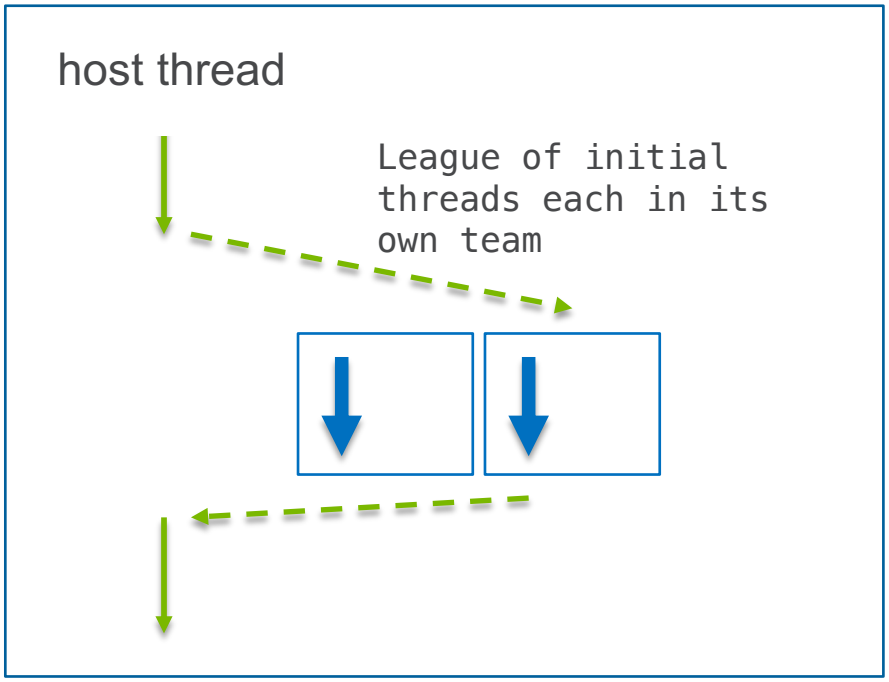
```
#pragma omp distribute [clause [[,]] clause ... ]
    for-loop
```

- Combined constructs

```
#pragma omp distribute parallel for[clause [[,]] clause ... ]
    for-loop
```

DISTRIBUTE CONSTRUCT

- Spreads the iterations of a loop coarsely across hardware compute units
  - Workshares by distributing iterations of a loop to the initial threads in a league.
  - Compare to **`for/do`** constructs, which assign work associated with loop iterations to threads inside a team




```
#pragma omp target teams num_teams(2)
#pragma omp distribute
  for(int i=0; i<N; i++ )
    {
      y[i] = x[i];
    }
```

DISTRIBUTE CONSTRUCT

```
#pragma omp target teams num_teams(2)
#pragma omp distribute
  for(int j=0; j<N; j+=N/2 )
    {
     #pragma omp parallel
     #pragma omp for
      for(int i=j; i< j+N/2; i++)
        y[i] = x[i];
    }
```

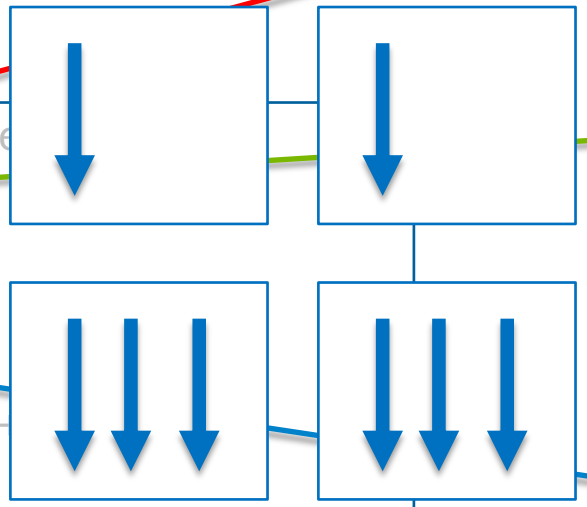
Creates a league of 2 teams, although only one thread per team is active

Distributes loop iterations across the master threads of each team

Activates the threads in the teams, and distributes the loop iterations to the threads

DISTRIBUTE CONSTRUCT

```
#pragma omp target teams num_te
#pragma omp distribute
  for(int j=0; j<N; j+=N/2 )
    {
     #pragma omp parallel
     #pragma omp for
      for(int i=j; i< j+N/2; i+
        y[i] = x[i];
    }
```



Creates a league of 2 teams, although only one thread per team is active

Distributes loop iterations across the master threads of each team

Activates the threads in the teams, and distributes the loop iterations to the threads

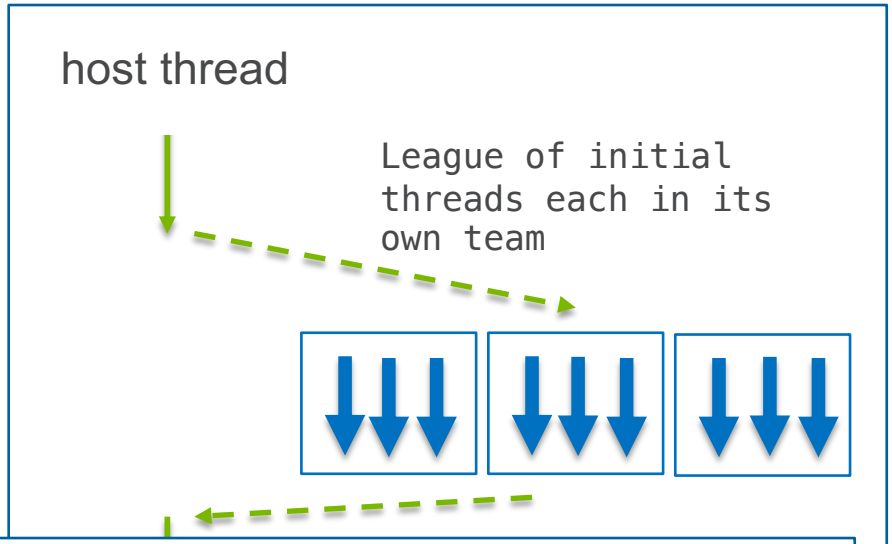
COMBINED CONSTRUCTS

- Purpose is to distribute loop iterations across multiple levels of parallelism without needing multiple loop nests

```
#pragma omp target teams distribute parallel for[clause[[,]clause] ] new-line
    for-loops
#pragma omp target teams distribute simd[clause[ [,] clause]] new-line
    for-loops
#pragma omp target teams distribute parallel for simd[clause[[,]…] ] new-line
    for-loops
Etc.
```

COMBINED CONSTRUCT

- Worksharing across two levels of parallelism using combined constructs




```
#pragma omp target teams distribute parallel for map(from:y[:N]) map(to:x[:N])
for(int i=0; i<N; i++ )
    {
      y[i] = x[i];
    }
```

Creates one thread on target device

Creates teams, although only the master thread is active

Distributes loop iterations across the initial thread of each team

Activates the threads in the teams, and distributes the loop iterations within the threads

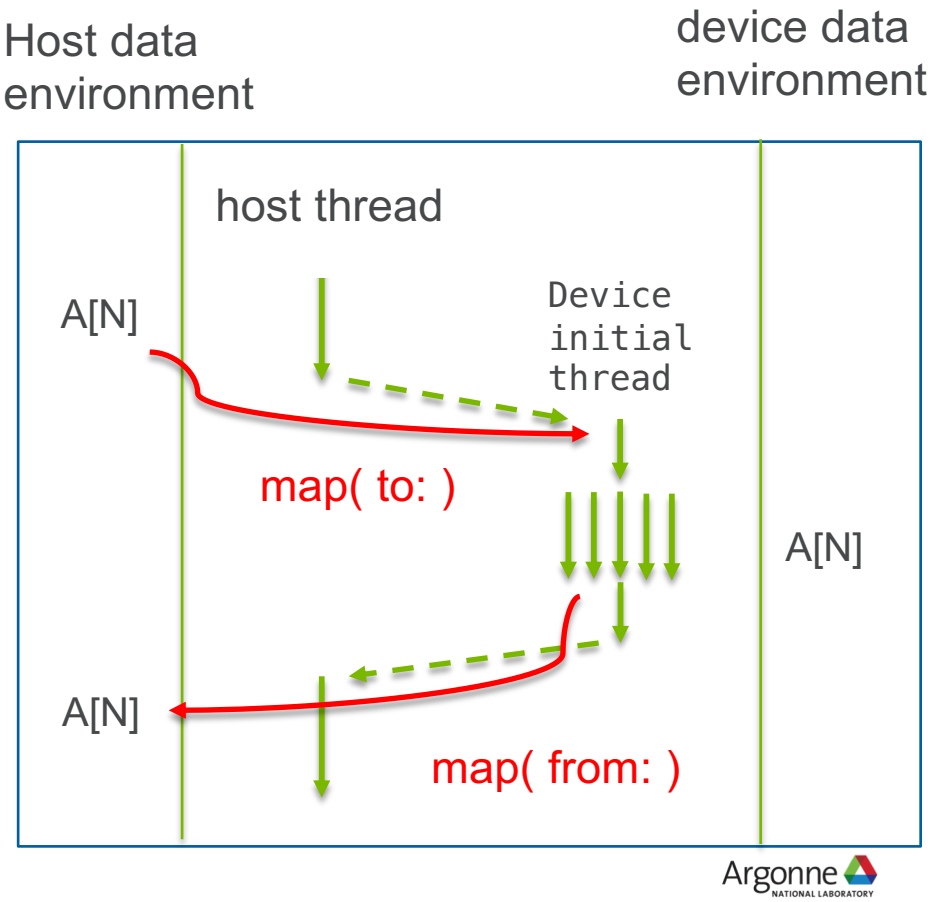
SUMMARY: DIFFERENCE BETWEEN TEAMS AND PARALLEL

- #pragma omp teams
  - Coarser-grained parallelism
  - Spawns multiple teams, each with one thread
  - (Typically) map to SMs in Nvidia HW
  - Threads in different teams can't synchronize with each other
  - **Distributes loop iterations with distribute**
- #pragma omp parallel
  - Finer-grained parallelism
  - Spawns many threads in a team
  - (Typically) map to CUDA cores in Nvidia HW
  - Threads in a team can synchronize with each other (#pragma omp barrier)
  - **Distributes loop iterations with for/do**

3. DATA MAPPING

DATA MAPPING CLAUSES

- Provides a mechanism for sharing variables between host and device
  - Memory may or may not be shared between host and device, but the implementation handles copying or not copying
- A mapped variable is the *corresponding variable* in a device data environment to the *original variable* in the host device environment
  - Can be thought of as a shared variable, distinct from (first)private variables



DATA MAPPING: IMPORTANT CONSTRUCTS

- Map clause on a target construct
  - Map variables for a single target region
  - Enclosed region executes on device and maps data

```
map([[map-type-modifier[,]]
map-type:] list)
```

- Target data
  - Map variables across multiple target regions in a structured block
  - Enclosed region does not execute on device, only maps data

```
#pragma omp target data clause
   structured block
```

- Declare target
  - Allows global variables to be mapped to an accelerator's device data environment for the whole execution of the program ("globally mapped").

```
#pragma omp declare target
   declarations-defs-seq
#pragma omp end declare target
```

- Target enter/exit
  - Map variables in stand-alone clauses

```
#pragma omp target enter data
clause [[[,] clause ... ]
```

DATA MAPPING: IMPORTANT CONSTRUCTS

- Map clause on a target construct
  - Map variables for a single target region
  - Enclosed region executes on device and maps data
  - Available *map-types* are:
    - map(to: X) : map to device before execution
    - map(from: X) : map from device after execution
    - map(tofrom: X) : map to/from device
    - map(alloc: X) : allocate on device
  - Defaults:
    - Arrays and structs are tofrom
    - Scalars are firstprivate
    - Be careful about pointers, since the memory address the pointer points to may not exist on the device

```
map([[map-type-modifier[,]]
map-type:] list)

Example:
#pragma omp target map(alloc:y)
```

if you don't map explicitly, the compiler will do it for you

MAP CLAUSE: IMPLICIT MAPPING

```
  int x[N];
  int y[N];
#pragma omp target

  {
    int i;
    for(i=0;i<N;i++)
      y[i] = i+N;


    for(i=0;i<N;i++)
      x[i] = y[i];
  }
```

4 data transfers

The default map-type for arrays is **tofrom**:

On entry to the target region, storage is allocated for arrays **y** and **x** on the device. Then the values of **x** and **y** are copied to the accelerator. (*map-entry*)

On exit from the target region, the accelerator's values of **x** and **y** are copied back to the host, and storage for **y** and **x** on the accelerator is released. (*map-exit*)

MAP CLAUSE: EXPLICIT MAPPING

```
  int x[N];
  int y[N];
#pragma omp target map(alloc:y) \
                   map(from:x)
  {
    int i;
    for(i=0;i<N;i++)
      y[i] = i+N;


    for(i=0;i<N;i++)
      x[i] = y[i];
  }
```

The default map-type for arrays is **tofrom**:

On entry to the target region, storage is allocated for arrays **y** and **x** on the device. The the values of **x** and **y** are left uninitialized (no copying).

On exit from the target region, the accelerator's value of **x** is copied back to the host, and storage for **y** and **x** on the accelerator is released

1 data transfer

DATA MAPPING: IMPORTANT CONSTRUCTS

- Target data
  - Map variables across multiple target regions in a structured block
  - Enclosed region does not execute on device, only maps data

```
#pragma omp target data clause
   structured block

Ex:
#pragma omp target data map(from: p[0:N])
  {
    #pragma omp target map(to: v1[:N], v2[:N])
    { }
    // host code
    #pragma omp target map(to: v1[:N], v2[:N])
    { }
  }
```

TARGET DATA CONSTRUCT: MAP VARIABLES ACROSS MULTIPLE TARGET REGIONS

```
int i;
init(v1, v2, N);


#pragma omp target map(to: v1[:N], v2[:N]) map(from: p[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

    init_again(v1, v2, N);

#pragma omp target map(to: v1[:N], v2[:N]) map(tofrom: p[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);

output(p, N);
```

1. On entry to first target region, **v1**, **v2**, **p** are allocated on the device. **v1** and **v2** are copied to the device.
2. On exit from the first target region, **p** is copied to host, and **v1**, **v2**, and **p** are removed from the device.
3. On entry to the second target region, **v1**, **v2**, **p** are allocated and copied.
4. Exit is the same as in the first target region
5. **p** is copied back and forth between the target regions, even though we don't modify the array between target regions

TARGET DATA CONSTRUCT: MAP VARIABLES ACROSS MULTIPLE TARGET REGIONS

```
  int i;
  init(v1, v2, N);
#pragma omp target data map(from: p[0:N])
  {
#pragma omp target map(to: v1[:N], v2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

    init_again(v1, v2, N);

#pragma omp target map(to: v1[:N], v2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);
  }
  output(p, N);
```

Array **p** is only mapped once, avoiding having to copy **p** back and forth

TARGET DATA CONSTRUCT: MAP VARIABLES ACROSS MULTIPLE TARGET REGIONS

```
  int i;
  init(v1, v2, N);
#pragma omp target data map(from: p[0:N])                    1
  {
#pragma omp target map(to: v1[:N], v2[:N])                   2
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];                                  3

    init_again(v1, v2, N);

#pragma omp target map(to: v1[:N], v2[:N])                   4
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);                         5
  }
  output(p, N);                                              6
```

1. **p** is new. alloc'd on the device and left uninitialized. Ref. count set to 1.
2. **v1**,**v2** are new. alloc'd on the device, ref. counts set to 1, host values copied to it. **p** is already present, ref count set to 2.
3. **v1**,**v2** ref counts decrement to 0, and are released. **p** ref count decremented to 1, not released.
4. Same as 2.
5. Same as 3.
6. Ref. count for **p** is 1, copied back to host, ref count decrements to 0, released.

TARGET DATA CONSTRUCT: MAP VARIABLES ACROSS MULTIPLE TARGET REGIONS

```
  int i;
  init(v1, v2, N);
#pragma omp target data map(from: p[0:N])
  {
#pragma omp target map(to: v1[:N], v2[:N]) map(from: p[0:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

    init_again(v1, v2, N);

#pragma omp target map(to: v1[:N], v2[:N]) map(tofrom: p[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);
  }
  output(p, N);
```

- There is only one instance of a variable in an accelerator's address space
- Map clause ignored if already mapped in an outer region

TARGET DATA CONSTRUCT: MAP VARIABLES ACROSS MULTIPLE TARGET REGIONS

```
  int i;
  init(v1, v2, N);
#pragma omp target data map(from: p[0:N])
  {
#pragma omp target map(to: v1[:N], v2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

    init_again(v1, v2, p, N);

#pragma omp target map(to: v1[:N], v2[:N]) \
                                map(always, to: p[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);
  }
  output(p, N);
```

- There is only one instance of a variable in an accelerator's address space
- Map clause ignored if already mapped in an outer region
- Can use "always" clause to force it

TARGET UPDATE CONSTRUCT: KEEPING HOST AND DEVICE CONSISTENT

```
  int i;
  init(v1, v2, N);
#pragma omp target data map(from: p[0:N])
  {
#pragma omp target map(to: v1[:N], v2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

    init_again(v1, v2, p, N);

#pragma omp target update to(p[:N])

#pragma omp target map(to:v1[:N], v2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      p[i] = p[i] + (v1[i] * v2[i]);
  }
  output(p, N);
```

- There is only one instance of a variable in an accelerator's address space
- Map clause ignored if already mapped in an outer region
- Can use "always" clause to force it
- Or can use **target update** to force it. When a host thread encounters a **target update** construct, it checks if the variables are present in the device data environment, and makes the consistent according to the to/from clause

DATA MAPPING: IMPORTANT CONSTRUCTS

- Declare target
  - Allows global variables to be mapped to an accelerator's device data environment for the whole execution of the program ("globally mapped").

```
#pragma omp declare target
   declarations-defs-seq
#pragma omp end declare target
```

DECLARE TARGET: GLOBAL VARIABLES TO DEVICE FOR PROGRAM LIFETIME

```
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target

void vec_mult()
{
   int i;
   init(v1, v2, N);
   #pragma omp target update to(v1, v2)

   #pragma omp target
   #pragma omp parallel for
   for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];

   #pragma omp target update from(p)

   output(p, N);
}
```

For variables, it allows global variables to be mapped to an accelerator's device data environment for the whole execution of the program ("globally mapped")

Globally mapped variables created in accelerator's device data environment at the start of the program, and have an infinite reference count (means that they are never copied to/from implicitly)

Target update is used to keep the variables consistent

```
class myArray {
public:
  int length;
  double *ptr;
  void allocate(int l) {
    double *p = new double[l];
    ptr = p;
    length = l;
#pragma omp target enter data \
                  map(alloc:p[0:length])
  }

  void release() {
    double *p = ptr;
#pragma omp target exit data \
                  map(release:p[0:length])
    delete[] p;
    ptr = 0;
    length = 0;
  }
};
```

Target enter/exit data

- **target enter data** construct executes a *map-enter* phase for the pointer-based array section **p[0:length]**. **alloc** allocates memory for the array section on the accelerator
- **target exit data** construct executes a *map-exit* phase for the pointer-based array section **p[0:length]**. Release frees the corresponding storage in the device data environment

SUMMARY OF DATA MAPPING CONSTRUCTS AND SCOPES

- Map clause on a target construct
  - Map variables for a single target region
  - Enclosed region executes on device and maps data
- Target data
  - Map variables across multiple target regions in a structured block
  - Enclosed region does not execute on device, only maps data
- Declare target
  - Allows global variables to be mapped to an accelerator's device data environment for the whole execution of the program ("globally mapped").
- Target enter/exit
  - Map variables in stand-alone clauses

Mapping is linked to the structured block inside the target data or target region

Maps a variable for the extent of the program, user-managed

Unstructured mapping, user-managed

SUMMARY: DECREASING DATA TRANSFER

- Use map clause to specify when an array/variable needs to be copied back and forth, instead of using default implicit tofrom clause
- Use target data regions around structured blocks to avoid mapping variables unnecessarily
- Use target enter/exit data and target declare data to manage data transfer more explicitly

4. RUNTIME ROUTINES (AND NVPROF)

RUNTIME ROUTINES

Helpful for understanding where execution is occurring

- Set default device
  - void **omp_set_default_device**(int num);
- Get default deviice
  - int **omp_get_default_device**();
- Get number of target devices
  - int **omp_get_num_devices**();
- Find out if we're on the host
  - int **omp_is_initial_device**();
- Find out who the host is
  - int **omp_get_initial_device**();
- Get information about the teams
  - int **omp_get_num_teams**();
  - int **omp_get_team_num**();

- OpenMP support multiple accelerators
- Devices each have a unique device number, although the number is implementation-defined
- **device** and **if** clauses determine the device
  - If not specified, *default-device-var* ICV is used (set to OMP_DEFAULT_DEVICE if set, otherwise it's implementation-defined)

ENVIRONMENT VARIABLES

Helpful for understanding where execution is occurring

- OMP_DEFAULT_DEVICE
  - set default device, when “device(num)” clause is not specified
- OMP_TARGET_OFFLOAD={"MANDATORY" | "DISABLED" | "DEFAULT" }
  - Controls whether region runs on device or host (OpenMP 5.0)

NVPROF: PROFILING TOOL FROM CUDA TOOLKIT

- Simple profiling: **`nvprof ./a.out`**

```
$ nvprof ./a.out
==89709== NVPROF is profiling process 89709, command: ./a.out
==89709== Profiling application: ./06_array_section
==89709== Profiling result:
Time(%)      Time  Calls  Avg       Min     Max     Name
99.97%  61.4535s  128  480.11ms  479.95ms  480.15ms  __omp_offloading_2d_a3d062f_main_l14
0.03%   18.550ms  129  143.79us  2.3680us  150.37us  [CUDA memcpy DtoH]
0.00%   1.3120us    1  1.3120us  1.3120us  1.3120us  [CUDA memcpy HtoD]
```

Timing for OpenMP target region

Timing for data transfer

NVPROF: PROFILING TOOL FROM CUDA TOOLKIT

- Simple trace profiling: **`nvprof --print-gpu-trace ./a.out`**

```
$ nvprof --print-gpu-trace ./a.out
==98180== NVPROF is profiling process 98180, command: ./a.out
Success!
==98180== Profiling application: ./06_map
==98180== Profiling result:
   Start  Duration  Grid Size Block Size  Regs*  Size   Name
374.62ms  2.3360us      -          -       -     1B     [CUDA memcpy DtoH]
374.69ms  1.3120us      -          -       -     4B     [CUDA memcpy HtoD]
374.98ms  2.6880us      -          -       -     4.0KB  [CUDA memcpy HtoD]
375.00ms  2.2080us      -          -       -     4.0KB  [CUDA memcpy HtoD]
375.07ms  3.4284ms   (1 1 1)  (128 1 1)    21     0B     __omp_offloading_2d_c1d8417
378.58ms  6.7840us      -          -       -     4.0KB  [CUDA memcpy DtoH]
378.66ms  6.7840us      -          -       -     4.0KB  [CUDA memcpy DtoH]
```

SUMMARY: CHECKING PROGRAM EXECUTION

- Use OpenMP runtime routines to check if you're running on the device, the number of threads/teams on the device
- Use nvprof to check the number of data transfers and where time is going

REFERENCES AND ACKNOWLEDGEMENTS

Material from the following were used in this presentation:

1. **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017
2. OpenMP 4.5 Specification and Examples documents
3. “OpenMP 4.5: Relevant Accelerator Features” https://www.olcf.ornl.gov/wp-content/uploads/2018/02/SummitDev_OpenMP4.5-tutorial-jan17.pdf
4. “Advanced OpenMP Tutorial” https://openmpcon.org/wp-content/uploads/openmpcon2017/Tutorial2-Advanced_OpenMP.pdf
5. “Targeting GPUs with OpenMP4.5 Device Directives” http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

OVERVIEW

- Introduction and some terminology
  - Execution model and data environment
- Important OpenMP 4.5 Constructs/Concepts
  - Device execution control
  - Workshare
  - Data mapping
  - Runtime routines
- Demo on JLSE at ALCF

DEMO

DEMO ON JLSE AT ALCF

*Examples can be run at other locations, too

1. Log into JLSE
   - ssh user@login.jlse.anl.gov
2. Get the examples
   - `git clone https://github.com/colleeneb/simple_offload_examples.git`

DEMO ON JLSE AT ALCF

*Examples can be run at other locations, too

3. Submit an interactive job, cd into the right directory, and set the environment

```
$ qsub -q gpu_v100_smx2 -n 1 -t 60 -I

$ cd location_of_cloned_git_repo/simple_offload_examples

$ source build_files/jlse/environment_setup.sh # sets paths
```

DEMO ON JLSE AT ALCF

*Examples can be run at other locations, too

4. Build the examples

```
$ make -f build_files/jlse/Makefile.jlse
clang++  -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda 01_target_construct.cpp -o
01_target_construct
clang++  -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda 02_target_teams.cpp -o
02_target_teams
clang++  -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda 03_target_teams_parallel.cpp
-o 03_target_teams_parallel
clang++  -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda 04_map.cpp -o 04_map
clang++  -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda 05_map_type.cpp -o
05_map_type
```

DEMO ON JLSE AT ALCF

*Examples can be run at other locations, too

1. “Hello world from the accelerator”
   - ./01_target_construct
2. Look at how many teams are generated with just `target teams` clause:
   - ./02_target_teams
   - This should print out a message from every thread in a team
3. Look at what happens when the `parallel` construct is added after `target teams`:
   - ./03_target_teams_parallel
4. Look at the number of data transfers printed out with nvprof when implicit mapping is used:
   - nvprof --print-gpu-trace ./04_map
5. Look at the number of data transfers printed out with nvprof when explicit mapping is used:
   - nvprof --print-gpu-trace ./05_map_type

BACKUP

MANAGING MEMORY WITH API ROUTINES

- Allocates space on the device, get a device pointer
  - void* **omp_target_alloc**(size_t size, int device_num)
- Free space on the device using a device pointer
  - void **omp_target_free**(void* device_ptr, int device_num);
- Copy memory back and forth
  - 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);