OAK RIDGE National Laboratory

Introduction to OpenMP Offload: Part I

September 29th , 2023

Swaroop Pophale (CSMD), Reuben Budiardja (NCCS), Wael Elwasif (CSMD)
Oak Ridge National Laboratory

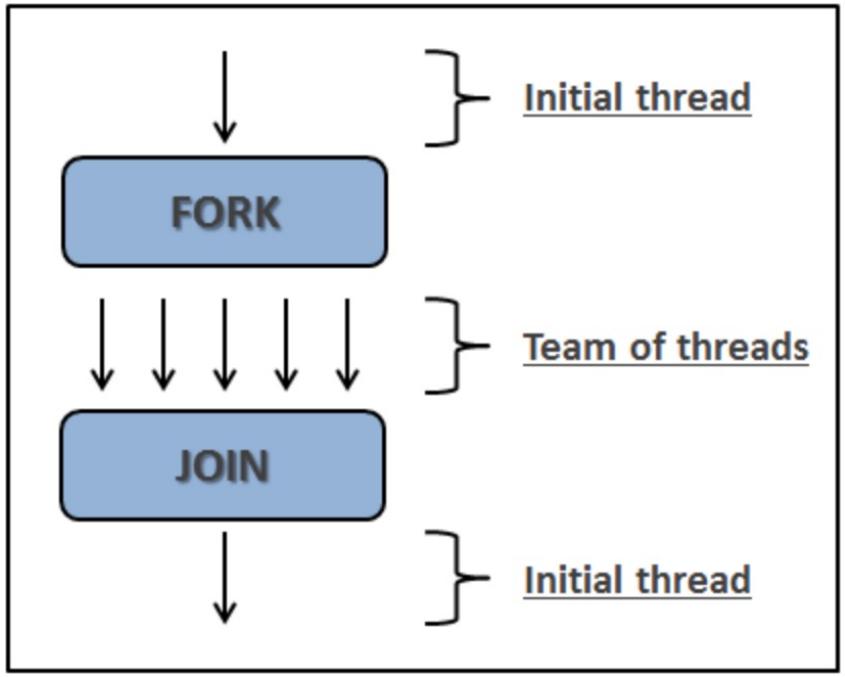
ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

OpenMP Programming Model

It is an Application Program Interface (API) to allow programmers to develop threaded parallel codes on shared memory computational units.

- Directives are understood by OpenMP aware compilers (others are free to ignore)
- Generates parallel threaded code
  - Original thread becomes thread “0”
  - Share resources of the original thread (or rank)
  - Data-sharing attributes of variables can be specified based on usage patterns



Reference: Somewhere from the web

OpenMP Worksharing

#pragma omp parallel → All threads will execute the region

#pragma omp parallel for → All threads will execute a part of the iterations

- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS env variable or num_threads clause (num_threads has precedence)

Recap: OpenMP Worksharing

Serial

```
for (int i = 0; i < N; ++i)
{
    C[i] = A[i] + B[i];
}
```

- 1 thread/process will execute each iteration sequentially
- Total time = time_for_single_iteration * N

Parallel

```
#pragma omp parallel
for (int i = 0; i < N; ++i)
{
    C[i] = A[i] + B[i];
}
```

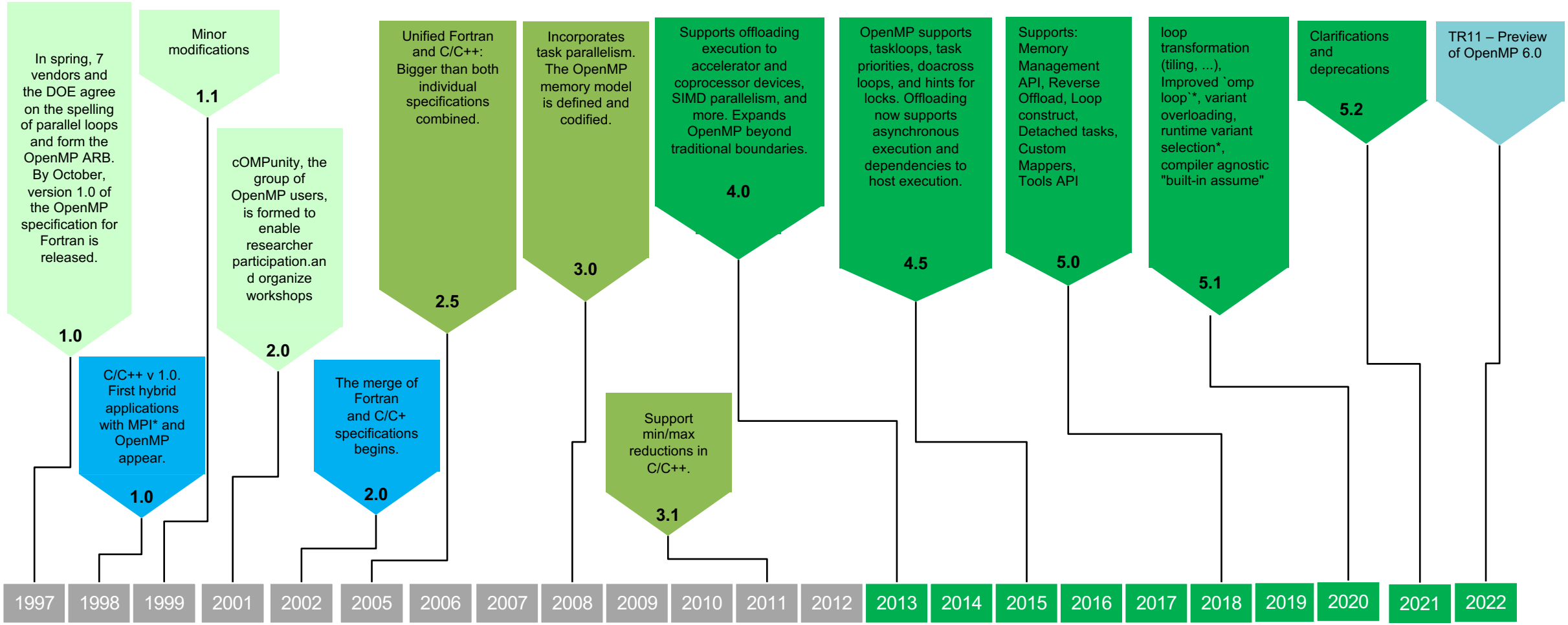
- Say, OMP_NUM_THREADS = 4
- 4 threads will execute each iteration sequentially (overwriting values of C)
- Total time = time_for_single_iteration * N

Parallel Worksharing

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
    C[i] = A[i] + B[i];
}
```

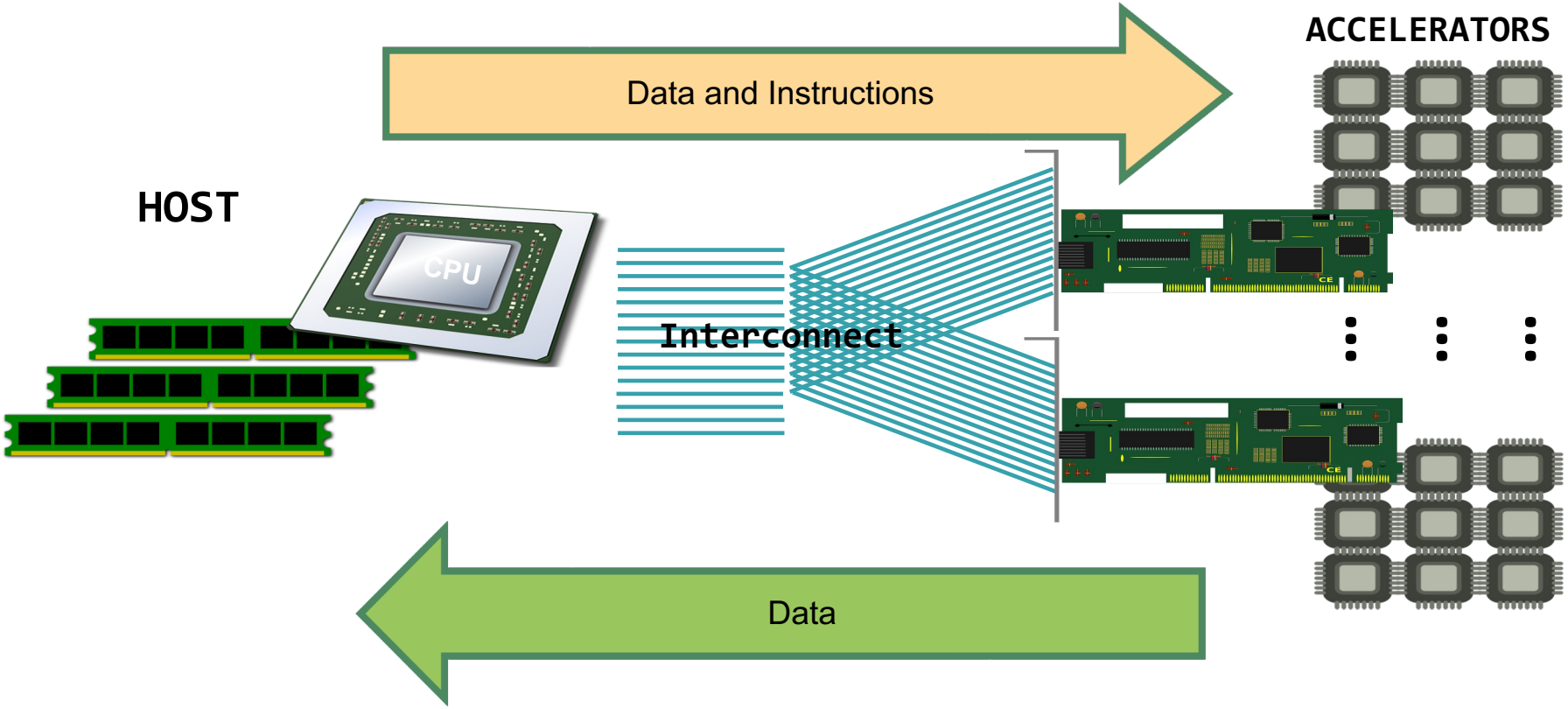
- Say, OMP_NUM_THREADS = 4
- 4 threads will distribute iteration space (roughly N/4 per thread)
- Total time = time_for_single_iteration * N/4

Evolution of OpenMP: 1997 – 2022



Introduction: OpenMP Offload

- OpenMP offload constructs are a set of directives for C, C++, and Fortran that were introduced in OpenMP 4.0 and further enhanced in later versions. Accelerators



OpenMP Offload Terminology

- **Host device**
  - The device on which the OpenMP program begins execution.
- **Target device**
  - A device with respect to which the current device performs an operation, as specified by a device construct or an OpenMP device memory routine.
- **Parent device**
  - For a given target region, the device on which the corresponding target construct was encountered.
    - A host device may not always be the parent.

OpenMP Offload: Steps

- **Identification** of compute kernels
  - CPU initiates kernel for execution on the device
- Expressing **parallelism** within the kernel
- Manage **data transfer** between CPU and Device
  - relevant data needs to be moved from host to device memory
  - kernel executes using device memory
  - relevant data needs to be moved from device to main memory

Step 1: Identification of Kernels to Offload

- Look for compute intensive code and that can benefit from parallel execution
  - Use performance analysis tools to find bottlenecks/computationally intensive kernels
- Track independent work units with well defined data accesses
- Keep an eye on platform specs
  - GPU memory is a precious resource
- Confirm via Profiling
  - Tools like rocprof and HPCToolkit

- More information regarding rocprof can be found at: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#optimization-and-profiling
- More information on HPCToolkit can be found at: http://hpctoolkit.org

How to Offload using OpenMP ?

C/C++	Fortran	Description
#pragma omp target *[clause[[,] clause] ...] new-line structured-block*	**!$omp target** *[clause[[,] clause] ...] loosely/tightly-structured-block* **!$omp end target**	The **target** construct offloads the enclosed code to the accelerator.

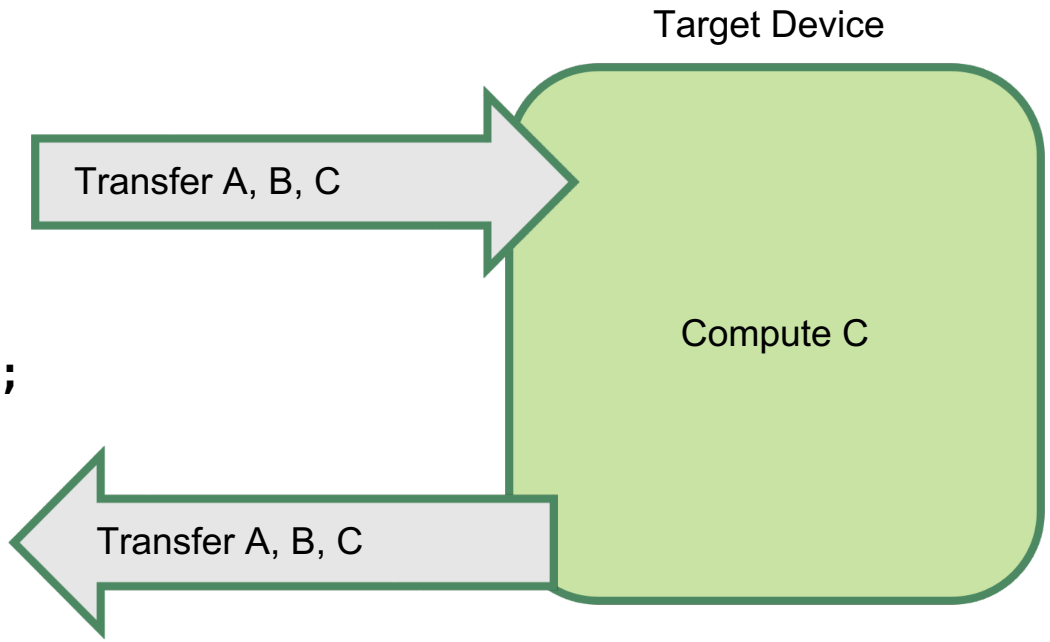
- A device data environment is created for the structured block
- The code region is mapped to the device and executed.

OpenMP Offload: Example using omp target

/*C code to offload Matrix Addition Code to Device*/

```
…
int A[N][N], B[N][N], C[N][N];
/*
  initialize arrays
*/
#pragma omp target
{
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      C[i][j] = A[i][j] + B[i][j];
    }
  }
} // end target
```

Kernel



The target construct is a task generating construct

Clauses on `target` directive

- Clauses allowed on the target directive:
  - device([device-modifier :] integer-expression)
  - if([target :] scalar-expression)
  - thread_limit(integer-expression)
  - private(list)
  - firstprivate(list)
  - in_reduction(reduction-identifier : list)
  - map([[map-type-modifier[,] [map-type-modifier[,] ...]] map-type:] locator-list)
  - is_device_ptr(list)
  - has_device_addr(list)
  - defaultmap(implicit-behavior[:variable-category])
  - nowait
  - depend([depend-modifier,] dependence-type : locator-list)
  - allocate([allocator :] list)
  - uses_allocators(allocator[(allocator-traits-array)] [,allocator[(allocator-traits-array)] ...])

`device` clause on `target` directive

- Use
  - Specify which device should execute the kernel
    - takes device_num or ancestor modifiers

/*C code depicting use of device clause */

```
#pragma omp target device(device_num:5) //same as device(5)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
  } // end target
```

Must be a valid device number

`device` clause to target multiple devices

```
/*C code to offload Matrix Addition Code to Multiple Devices*/
```

```
  …
int num_dev = omp_get_num_devices();
/*
Calculate start array index for each device and elements per device
*/

for (int dev = 0; dev < num_dev; ++dev)
{
 #pragma omp target map(tofrom: C[lb:len:1]) device(dev)
 {
   for (int i = lb; i < lb+len; ++i) {
     C[i] += A[i] + B[i] ;
    }
  } // end of omp target
}//end-for
```

device clause on target directive (cont.)

- Using the ancestor modifier

/*C code depicting use of device clause with ancestor modifier */

```
#pragma omp requires reverse_offload //at filescope
…
…

#pragma omp target
{

   #pragma omp target device(ancestor: 1)

    /*some useful work on parent device*/

   /* Continue with device execution*/

}//end target-1
…
```

`if` clause on `target` directive

- Use
  - Conditional execution on target device

/*C code demonstrating conditional offloading */

```
#pragma omp target if (N > 1024)

{
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {

        C[i][j] = A[i][j] + B[i][j];

      }
    }

} // end target
```

`nowait` clause on `target` directive

- Use
  - target task may be deferred

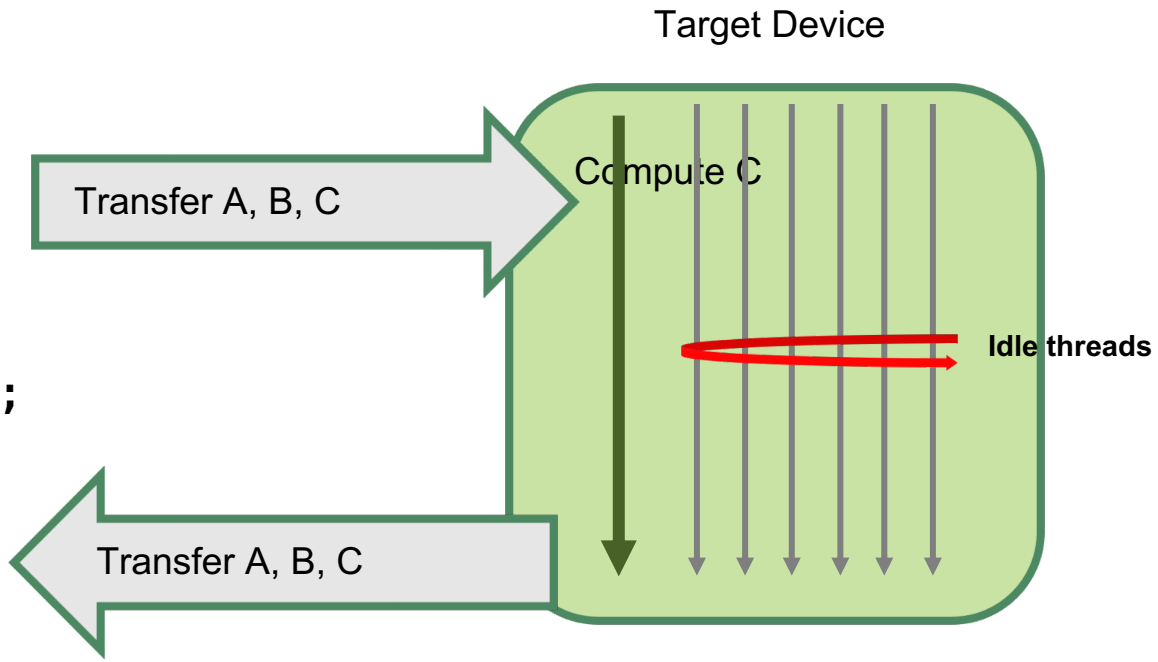
/*C code with nowait on target */
/*Use-case: Free host thread*/

```
…
#pragma omp parallel
{
  #pragma omp masked
  #pragma omp target nowait
    {
     /*independent work unit*/
    } // end target
  #pragma omp for
  for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B [i]
  }
  } // end parallel
```

Step 2: Expressing Parallelism

/*C code to offload Matrix Addition Code to Device*/

```
  …
  int A[N][N], B[N][N], C[N][N];
  /*
    initialize arrays
  */
#pragma omp target
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
  } // end target
```



Expressing Parallelism: using combined constructs

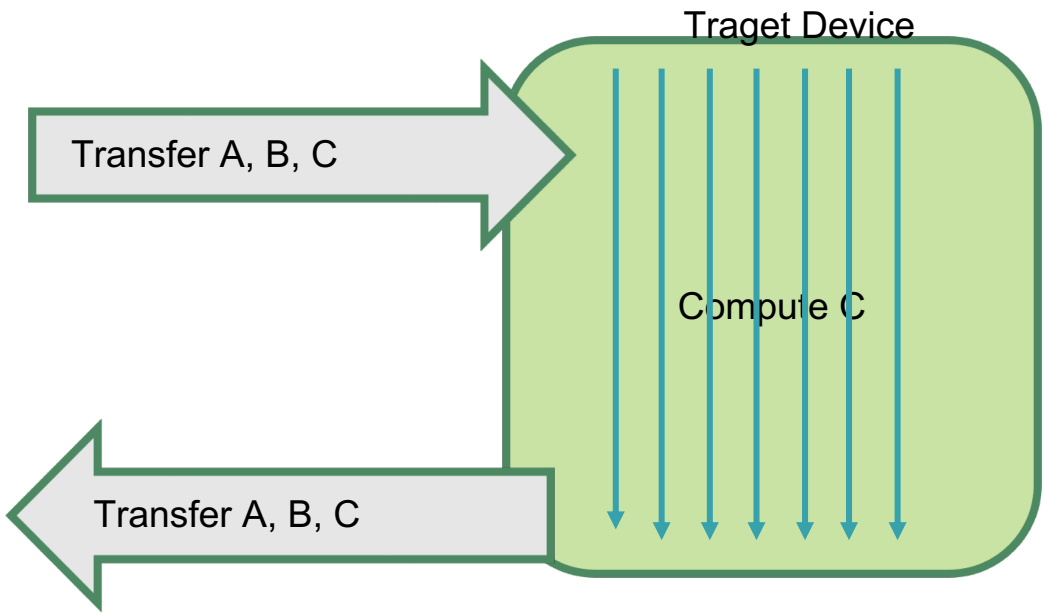
- Combined construct
  - A construct that is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

- Example:
  - `omp parallel{omp for } == omp parallel for`

target + parallel construct

/*C code using target parallel*/

```
  …
  int A[N], B[N], C[N];
  /*
    initialize arrays
  */
#pragma omp target parallel for
for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B[i];
 }
```



OpenMP `teams`

- When a `teams` construct is encountered, a league of teams is created.
- Each team is an initial team, and the initial thread “0” in each team executes the teams region.
  - Initial team numbers are consecutive whole numbers (zero to one less than the number of initial teams)
- The number of teams created is determined by the `num_teams` clause. Once the teams are created, the
  - these remain constant for the duration of the teams region.
- The `teams` region must be strictly nested within:
  - the implicit parallel region that surrounds the whole OpenMP program or
  - a target region.

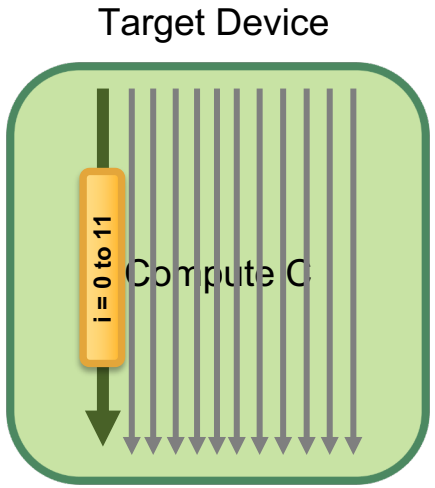
`distribute` construct

- It is a loop associated construct that binds to the set of initial threads executing an enclosing teams region
  - distribute construct must be strictly nested inside a teams region
- The iterations are distributed across the initial threads of all initial teams that execute the teams region to which the distribute region binds
- Clauses permitted on distribute construct are allocate, collapse, dist_schedule, firstprivate, lastprivate, order, and private

Expressing Parallelism: Increasing device utilization

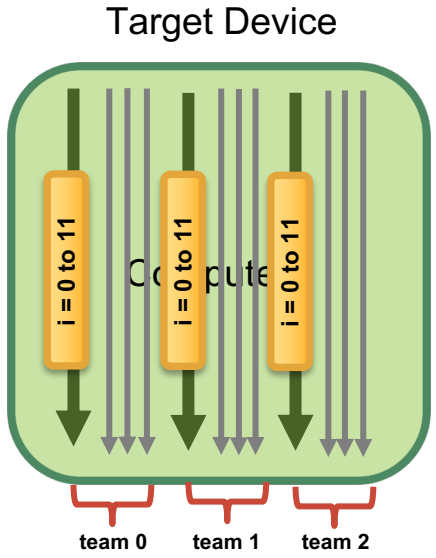
target

```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



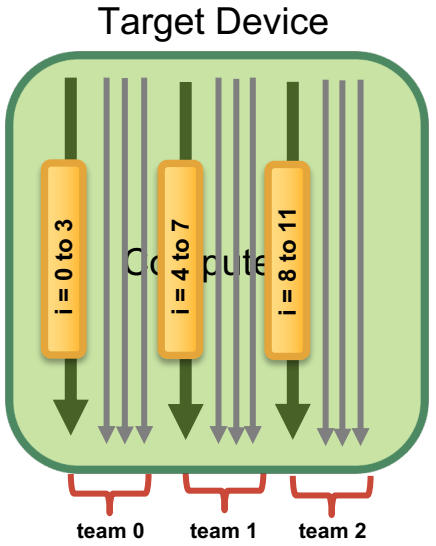
target teams

```
#pragma omp target teams
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



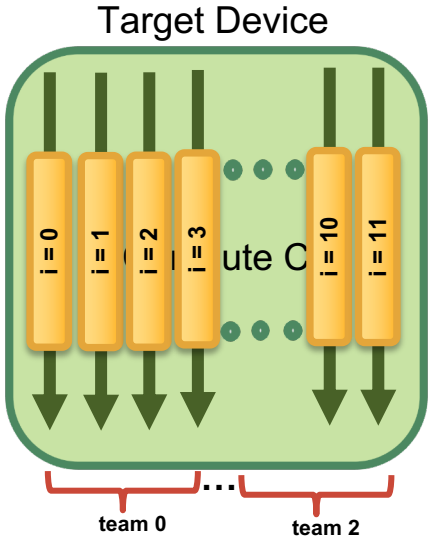
target teams distribute

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



target teams distribute parallel

```
#pragma omp target teams
distribute parallel
for[simd] num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



`loop` construct

- Properties:
  - logical iterations of the associated loops may execute concurrently
  - bind clause determines the binding region
    - orphaned loop needs explicit binding
  - can be nested inside another `loop` construct
  - all iterations are guaranteed to complete at the end of `loop`
    - except when bound to `teams` construct

Offloading using target teams + loop

/*C code with loop enclosed by teams region */

```
…
#pragma omp target teams
{
  #pragma omp loop //implicit bind(team)
  for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B [i]
  }
} // end target teams
```

/*C code with orphaned loop */

```
…
void fun1(){
#pragma omp loop bind(teams)
for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B [i]
}
#pragma omp target teams
{
  fun1();

} // end target teams
```

Summary: Device Execution Directives

C/C++	Fortran	Description
#pragma omp target *[clause[[,] clause] ...] new-line* *structured-block*	**!$omp target** *[clause[[,] clause] ...]* *loosely/tightly-structured-block* **!$omp end target**	The **target** construct offloads the enclosed code to the accelerator.
#pragma omp target teams *[clause[[,] clause] ...] new-line* *structured-block*	**!$omp target teams** *[clause[[,] clause] ...]* *loosely/tightly-structured-block* **!$omp end target teams**	The **target** construct offloads the enclosed code to the accelerator. The **teams** construct creates a league of teams. The **initial** thread of each team executes the code region.
#pragma omp target teams distribute *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target teams distribute** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target teams distribute***]*	The **target** construct offloads the enclosed code to the accelerator. A **league** of thread teams is created, and loop iterations are **distributed** and executed by the initial teams.
#pragma omp target teams distribute parallel for *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target teams distribute parallel do** [clause[[,] clause] ...] loop-nest *[***!$omp end target teams distribute parallel do***]*	The **target** construct offloads the enclosed code to the accelerator. A **league** of thread teams are created, and loop iterations are **distributed** and **executed in parallel** by all threads of the teams.

Summary: Device Execution Directives

C/C++	Fortran	Description
#pragma omp target parallel *[clause[[,] clause] ...]* new-line structured-block	**!$omp target parallel** *[clause[[,] clause] ...]* *loosely-structured-block* **!$omp end target parallel**	The **target** construct offloads the enclosed code to the accelerator. The **parallel** construct creates a team of OpenMP threads that execute the region.
#pragma omp target parallel for *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target parallel do** *[clause[[,] clause] ...]* *loop-nest* [**!$omp end target parallel do**]	The **target** construct offloads the enclosed code to the accelerator. The **parallel for/do** combined construct creates a thread team and distributes the inner loop iterations over threads.
#pragma omp target parallel loop *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target parallel loop** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target parallel loop***]*	The **target** construct offloads the enclosed code to the accelerator. The **parallel** construct creates a team of OpenMP threads that execute the region. The **loop** construct allows concurrent execution of the associated loops.
#pragma omp target teams loop *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target teams loop** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target teams loop***]*	The **target** construct offloads the enclosed code to the accelerator. The **teams** construct creates a league of teams. The **loop** construct allows concurrent execution of the associated loops.

Summary: Device Execution Directives (SIMD)

C/C++	Fortran	Description
#pragma omp target simd *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target simd** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target simd***]*	Semantics are identical to explicitly specifying a target directive immediately followed by SIMD directive.
#pragma omp target parallel for simd \ *clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target parallel do simd** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target parallel do simd***]*	Semantics are identical to explicitly specifying a target directive immediately followed by a parallel worksharing-loop SIMD directive.
#pragma omp target teams distribute simd \ *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target teams distribute simd** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target teams distribute simd***]*	Semantics are identical to explicitly specifying a target directive immediately followed by a teams distribute simd directive
**#pragma omp target teams distribute parallel for simd ** *[clause[[,] clause] ...] new-line* *loop-nest*	**!$omp target teams distribute parallel do simd** *[clause[[,] clause] ...]* *loop-nest* *[***!$omp end target teams distribute parallel do simd***]*	Semantics are identical to explicitly specifying a target directive immediately followed by a teams distribute parallel worksharing-loop SIMD directive.

Summary: Useful runtime routines for device environment

C/C++	Fortran	Where to call ?		Description
		Host	Target region	
int omp_get_num_procs(void);	integer function omp_get_num_procs()	✓	✓	returns the number of processors available to the device
void omp_set_default_device(int device_num);	subroutine omp_set_default_device(device_num) integer device_num	✓	✗	sets the value of the default-device-var ICV of the current task to device_num
int omp_get_default_device(void);	integer function omp_get_default_device()	✓	✗	returns the default target device
int omp_get_num_devices(void);	integer function omp_get_num_devices()	✓	✗	returns the number of non-host devices available for offloading code or data.
int omp_get_device_num(void);	integer function omp_get_device_num()	✓	✓	returns the device number of the device on which the calling thread is executing
int omp_is_initial_device(void);	logical function omp_is_initial_device()	✓	✓	returns true if the current task is executing on the host otherwise, it returns false.
int omp_get_initial_device(void);	integer function omp_get_initial_device()	✓	✗	return the device number of the host device

Summary: Useful runtime routines for teams region

C/C++	Fortran	Where to call ?		Description
		Host	Target region	
int omp_get_num_teams(void);	integer function omp_get_num_teams()	✓	✓	returns the number of initial teams in the current teams region.
int omp_get_team_num(void);	integer function omp_get_team_num()	✓	✓	returns the initial team number of the calling thread
void omp_set_num_teams(int num_teams);	subroutine omp_set_num_teams(num_teams) integer num_teams	✓	✓	the number of threads to be used for subsequent teams regions that do not specify a num_teams clause
int omp_get_max_teams(void);	integer function omp_get_max_teams()	✓	✓	returns an upper bound on the number of teams that could be created by a teams construct
void omp_set_teams_thread_limit(int thread_limit);	subroutine omp_set_teams_thread_limit(thread_limit) integer thread_limit	✓	✓	defines the maximum number of OpenMP threads per team

References

- Examples were adapted from:
  - https://github.com/SOLLVE/sollve_vv
  - OpenMP Examples Document 5.2.1
- OpenMP Specification (5.x)
  - https://www.openmp.org/specifications/
- https://www.nas.nasa.gov/hecc/assets/pdf/training/OpenMP4.5_3-20-19.pdf
- OpenMP Disussion @ 2021 Exascale Computing Project Virtual Annual Meeting (April 12 – 16, 2021)

Thank You