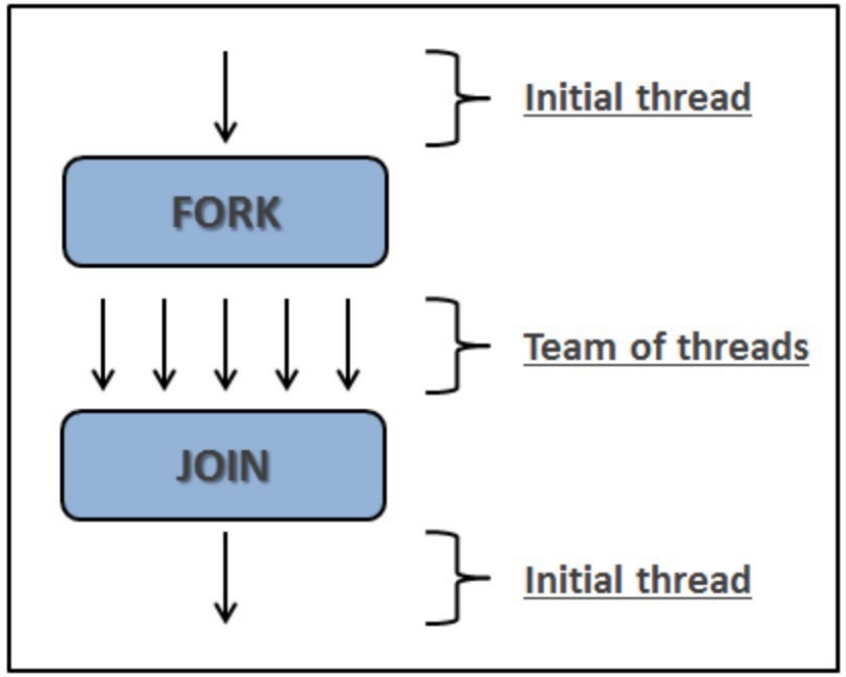
GPGPU Programming using OpenMP

3rd Latin American Introductory School on Parallel Programming and Parallel Architecture for HPC

Introduction to OpenMP

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



Recap: 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

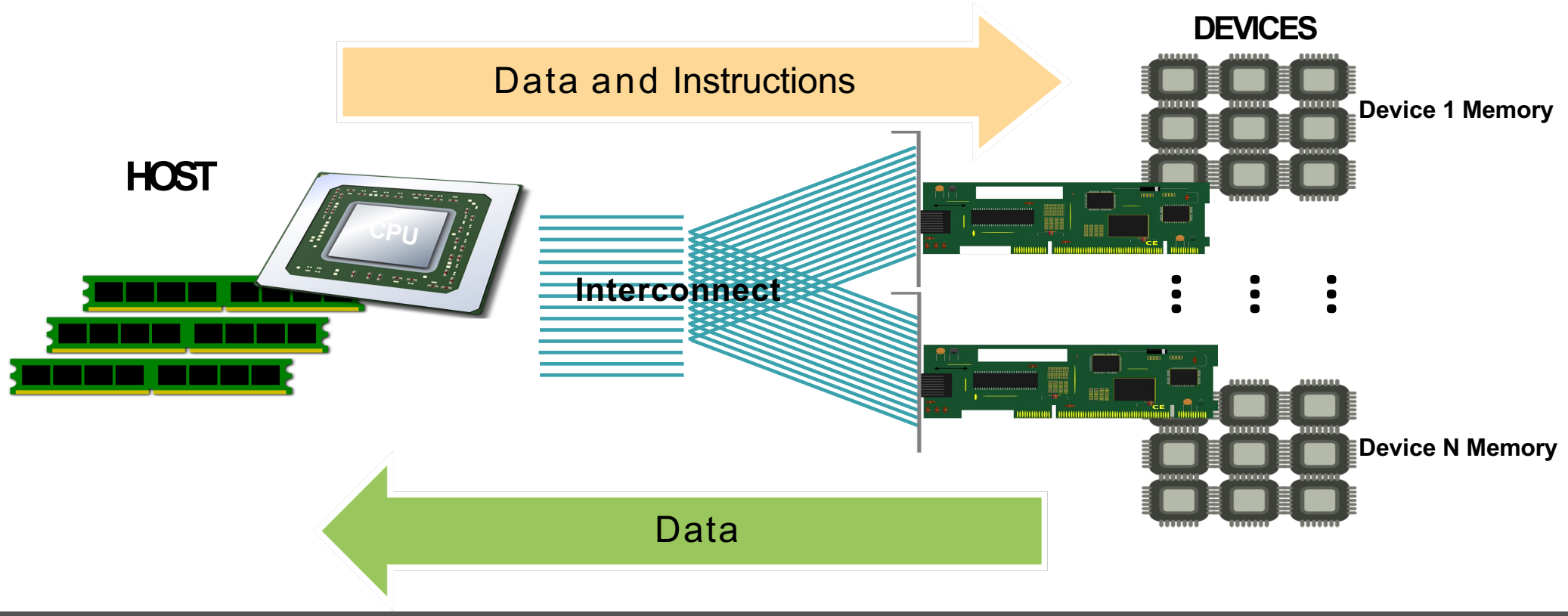
Parallel Worksharing

```
#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

Introduction: OpenMP Offload

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



OpenMP Offload: Steps

1. **Identification** of compute kernels
    - CPU initiates kernel for execution on the device
2. 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
3. Expressing **parallelism** within the kernel

Step 1: Identification of Kernels to Offload

- Look for compute-intensive code that can benefit from parallel execution
  - Use performance analysis tools to find bottlenecks
- Track independent work units with well-defined data access
- Keep an eye on platform specs
  - GPU memory is a precious resource

How to Offload ?

C/C++ API	Fortran API	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.

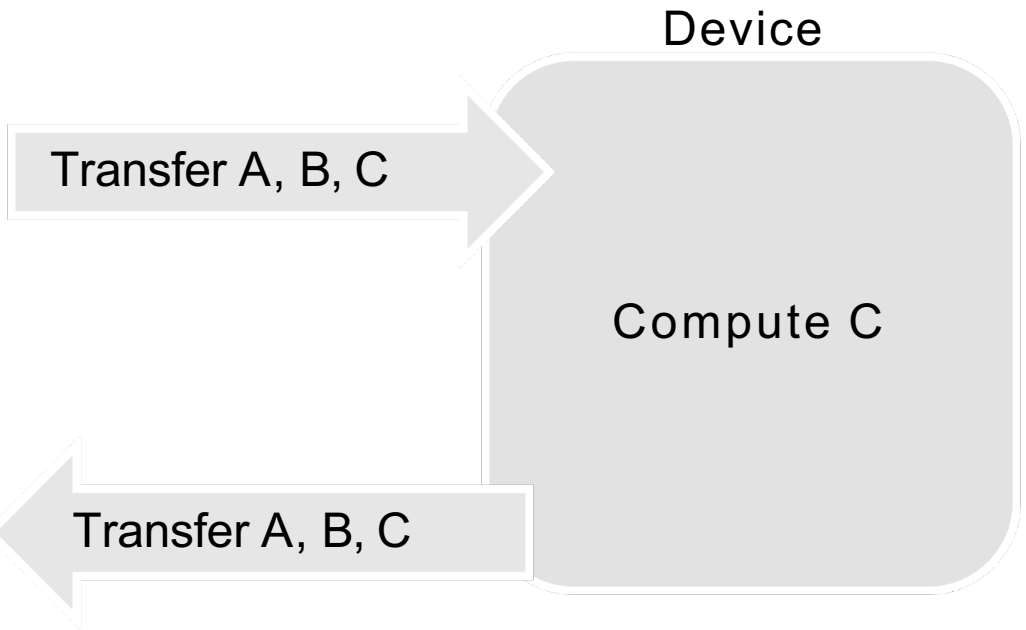
- A device data environment is created for the structured block
- The code region is mapped to the device and executed.

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
```

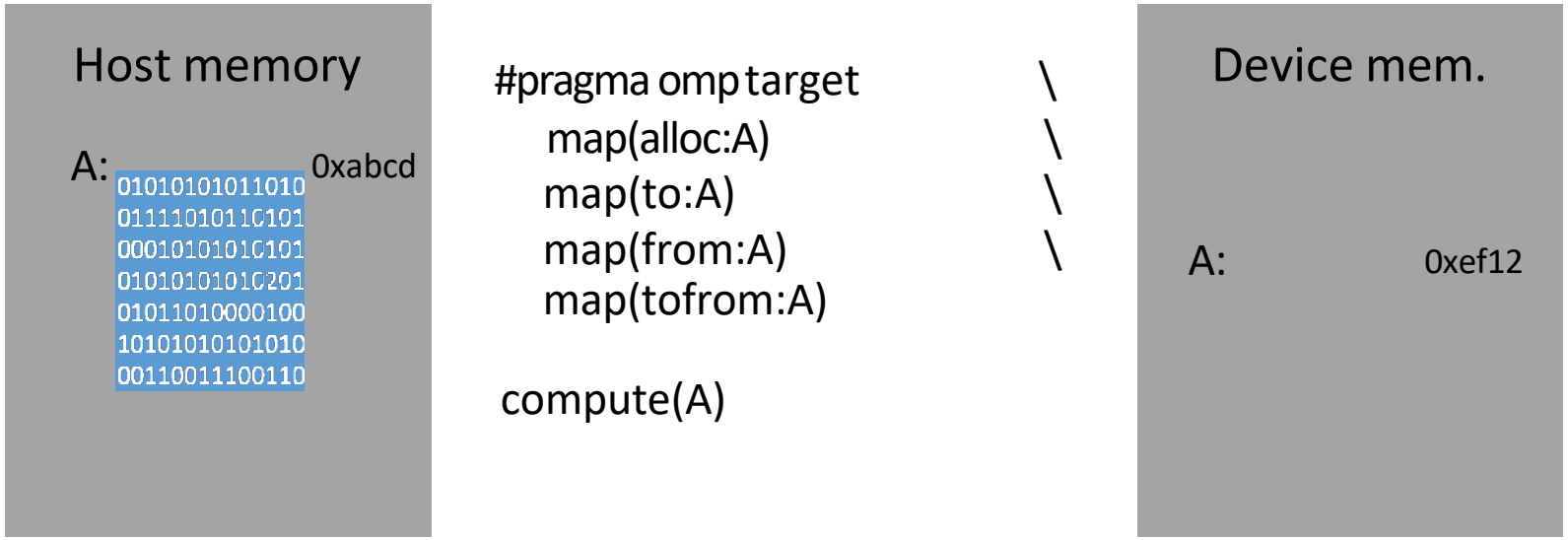
Kernel



The target construct is a task generating construct

Step 2: Manage Data Transfers

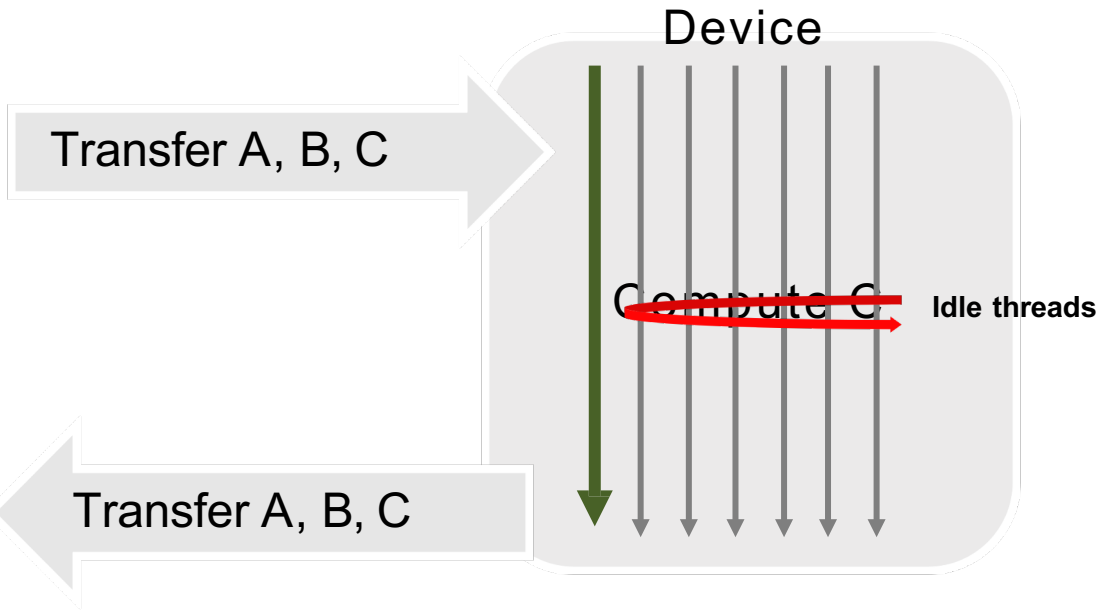
- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created for the structured block
  - Data environment is automatically destroyed at the end of the scope
  - Data transfers (if needed) are done at target invocation, too:
    - Upload data from the host to the target device.
    - Download data from the target device.



Step 3: 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 map(to:A, B, C)
  {
    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
```



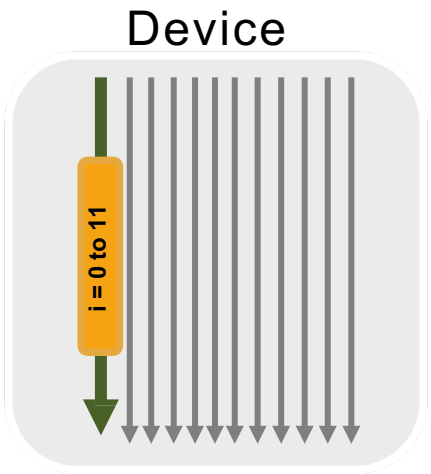
Expressing Parallelism: Device Execution Directives

C/C++ API	Fortran API	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.

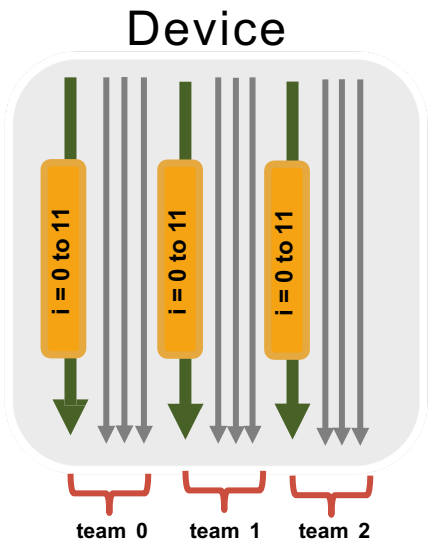
Expressing Parallelism: Increasing device utilization

target	target teams	target teams distribute	target teams distribute parallel

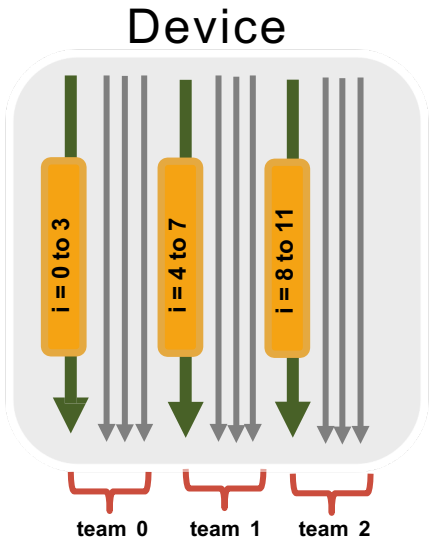
```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



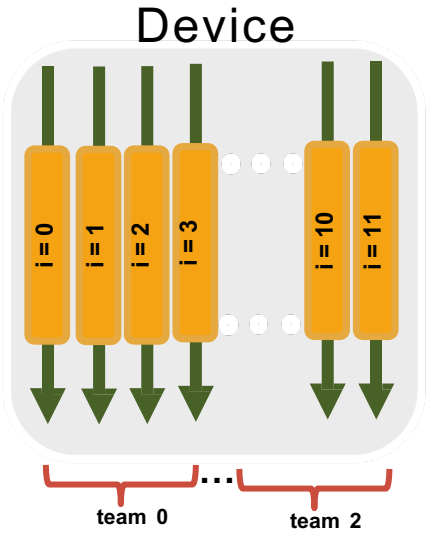

```
#pragma omp target teams
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```




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




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



Expressing Parallelism: 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 a 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.

Expressing Parallelism: Multiple devices

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

```
  …
int num_dev = omp_get_num_devices();
/*
Calculate the 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
```

Useful RT Routines: Device Environment

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

Teams Region: Useful RT Routines

host and target region

C/C++	Description
int **omp_get_num_teams**(void);	returns the number of initial teams in the current teams region.
int **omp_get_team_num**(void);	returns the initial team number of the calling thread
void **omp_set_num_teams**(int 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);	returns an upper bound on the number of teams that could be created by a teams construct
void **omp_set_teams_thread_limit**(i nt thread_limit);	defines the maximum number of OpenMP threads per team

References

- Examples were adapted from: https://github.com/SOLLVE/sollve_vv
- OpenMP Specification 5.1
- https://www.nas.nasa.gov/hecc/assets/pdf/training/OpenMP4.5_3-20-19.pdf