General Interest Seminar

Introduction to GPU programming with OpenMP

Jemmy Hu

SHARCNET HPC Consultant

January 31, 2024

OpenMP overview

OpenMP: A popular, portable and widely supported shared-memory parallel programming model in HPC

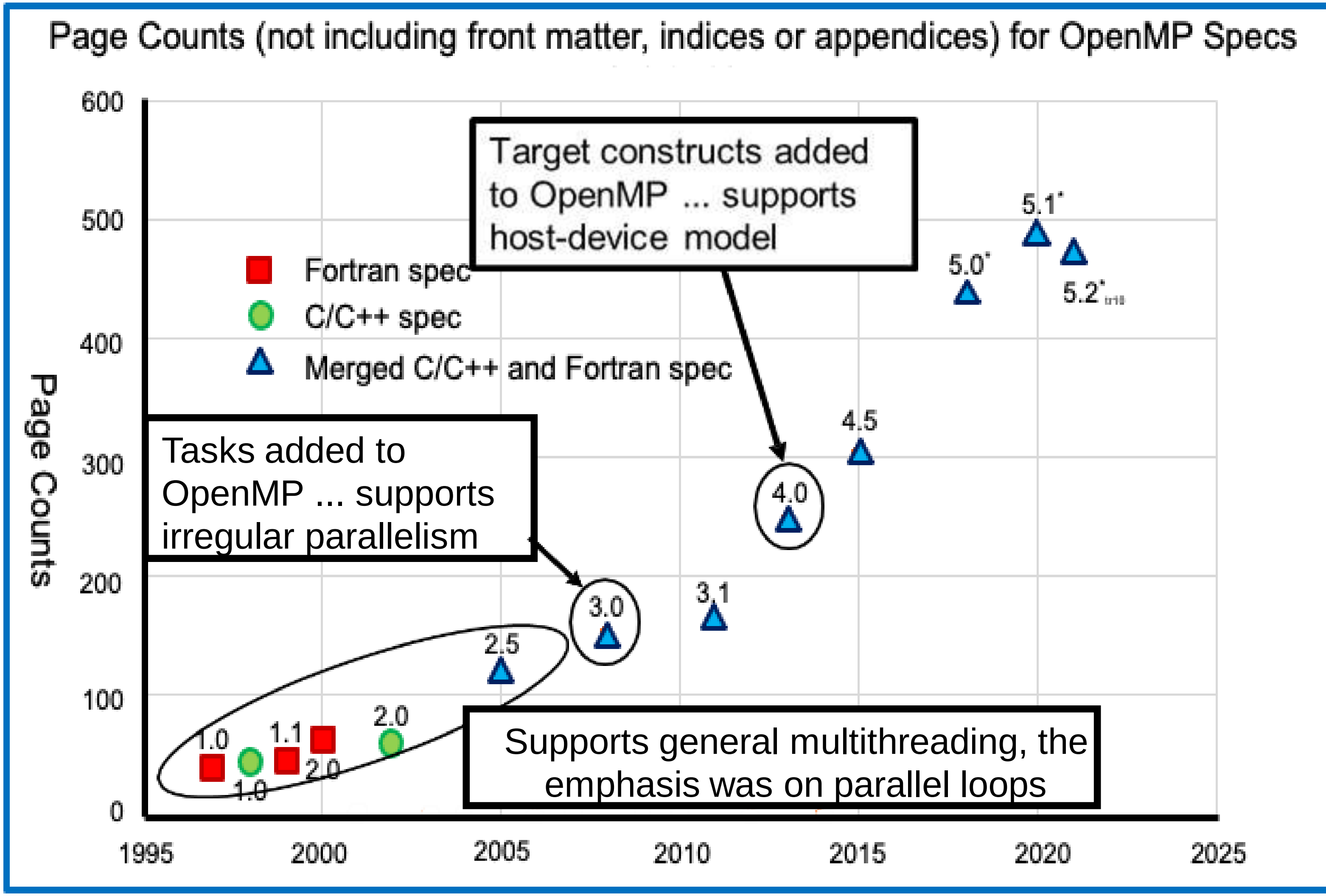
§ OpenMP API includes a set of compiler directives, library routines, and environment variables for parallel application programming

§ Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++

§ Ease of Use: Provide capability to incrementally parallelize a serial program, unlike message-passing libraries (MPI) which typically require an all or nothing approach

§ Standardizes established SMP practice + vectorization and heterogeneous device programming

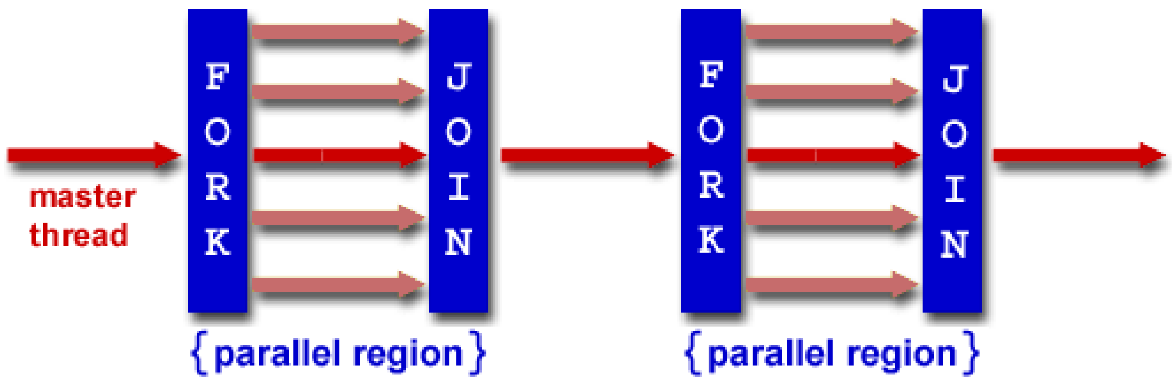
OpenMP History: The growth of complexity



OpenMP: Fork-Join Model

When OpenMP was originally launched, the focus was on Symmetric Multiprocessing, i.e. lots of threads with "equal access" to memory

- OpenMP uses the fork-join model of parallel execution:



FORK: the master thread then creates a ***team*** of parallel threads
The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Loop Parallelism

Sequential code	`for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}`
OpenMP parallel region	see code below
OpenMP parallel region and a work-sharing for-construct	see code below

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

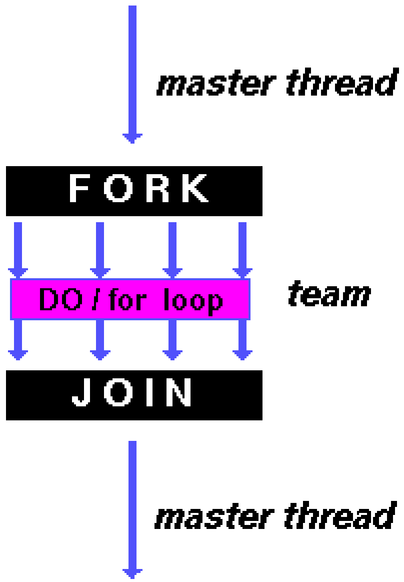
```
#pragma omp parallel
#pragma omp for schedule(static)
        for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

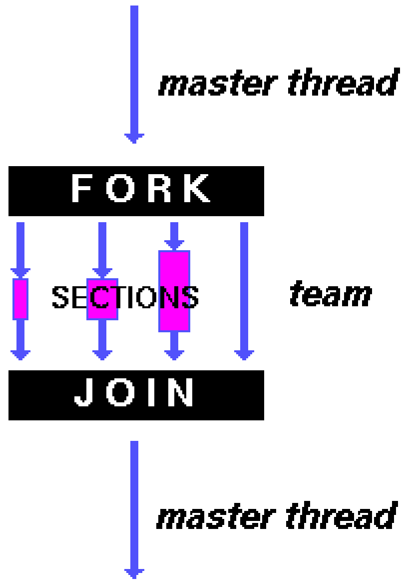
Types of Work-Sharing Constructs (Past):

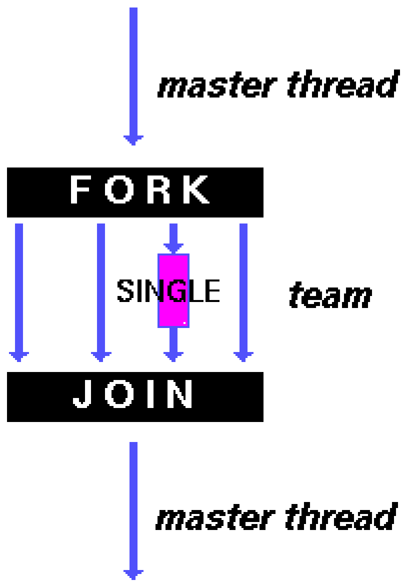
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

SINGLE - serializes a section of code



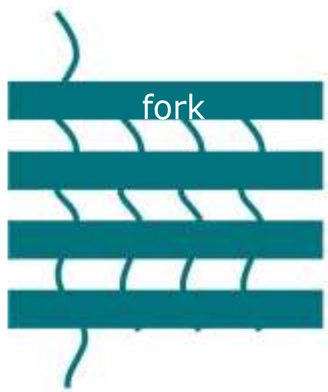




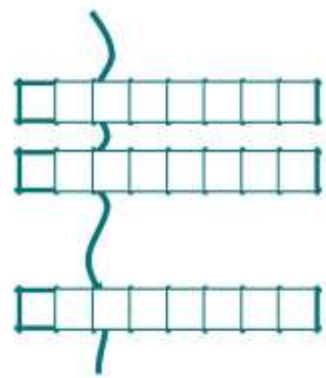
Existing Parallel Loop Constructs

- Existing parallel loop constructs are tightly bound to execution model:

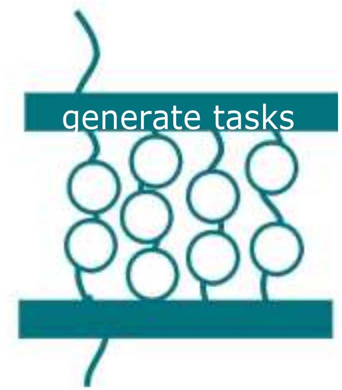
```
#pragma omp for
for (i=0; i<N;++i) {…}
```




```
#pragma omp simd
for (i=0; i<N;++i) {…}
```



```
#pragma omp taskloop
for (i=0; i<N;++i) {…}
```



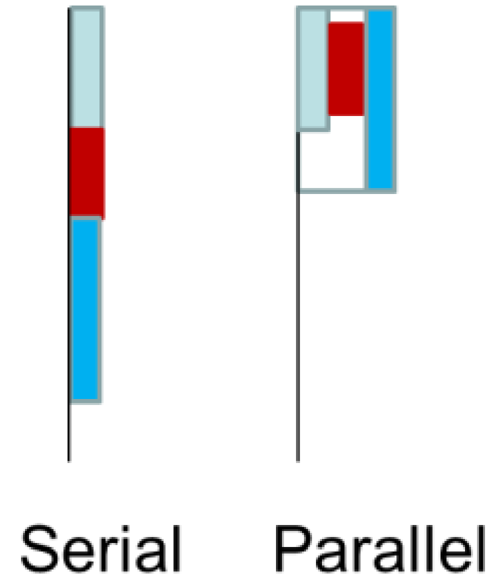
Not all programs have simple loops OpenMP can parallelize

- Consider a program to traverse a linked list:

```
p=head;
while (p) {
        processwork(p);
        p = p->next;
}
```

- OpenMP can only parallelize loops in the basic standard form with loop counts known at runtime

Task constructs in OpenMP



• The task construct was added to support irregular programs:

– While loops or loops whose iteration limits are not known at compiler time.

– Recursive algorithms

– divide and conquer problems.

• The task construct has expanded over the years with new features to support irregular problems with tasks in each new release of OpenMP

#pragma omp task

- Creates a new task, Task added to task queue
- Available thread picks next task from queue to execute

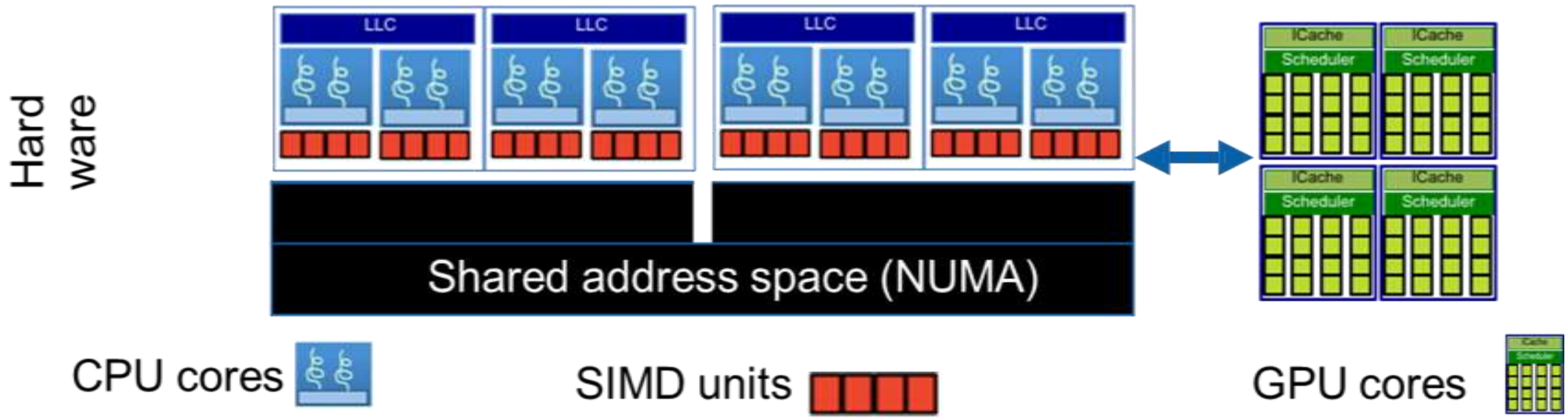
#pragma omp taskwait

- Acts like barrier
- Waits until all child tasks have finished

Linked lists with tasks

```
#pragma omp parallel
{
   #pragma omp single
   {
      p=head;
      while (p) {
         #pragma omp task firstprivate(p)
              processwork(p);
         p = p->next;
      }
    }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined



GPUs are made of many cores (compute units)

NVIDIA V100 has 80 Streaming Multiprocessors (SMs); these are the compute units
NVIDIA A100 has 108 compute units
Each NVIDIA compute unit has 64 FP32 processing elements
GPUs from AMD have similar structure of compute units and processing elements

On an A100, that’s 108 x 64 = 6,912 processing elements available to work in parallel

OpenMP for Accelerators: host/device Model

- **Host-centric: the execution of an OpenMP program starts on the *host device* and it may offload *target regions* to *target devices***

 In principle, a target region also begins as a single thread of execution: when a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread/task [on the host] waits at the construct until the execution of the region completes

 If a target device is not present, or not supported, or not available, the target region is executed by the host device

 If a construct creates a *data environment*, the data environment is created at the time the construct is encountered

Target Construct and data environment

- There are distinct memory spaces on host and device.
- OpenMP uses a combination of *implicit* and *explicit* data movement.
- Data may move between the host and the device in well defined places:
  - Firstly, at the beginning and end of a **target** region:

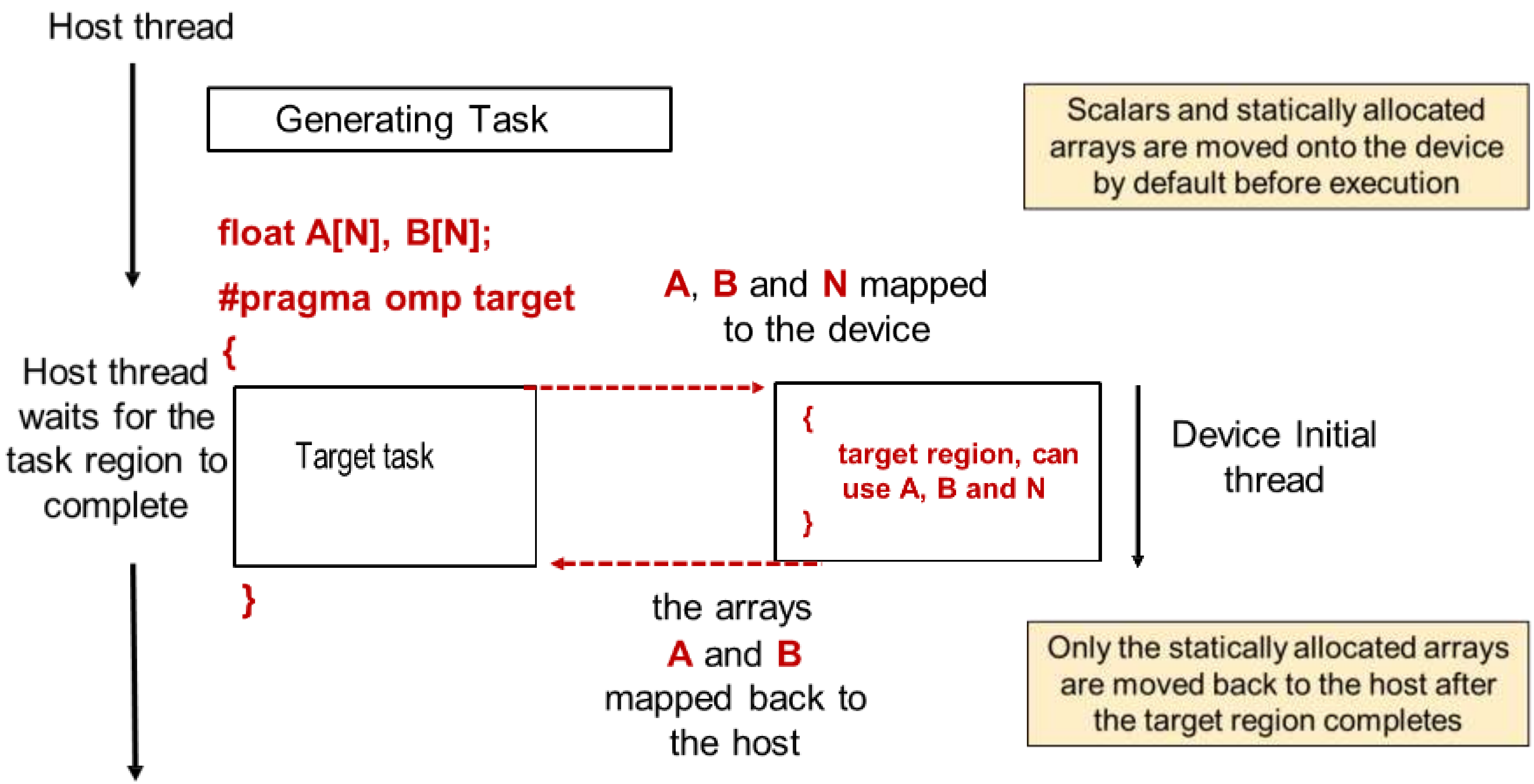
```
#pragma omp target
{       // Data may move from host to device here

 …

        // and from device to host here
 }
```

Host (CPU) – Device (GPU)

The target construct offloads to a device with two roles:
- transfer execution to the device
- transfer data to/from the device



Data environment

- **When an OpenMP program begins, each device has an initial *device data environment***

- **Directives accepting data-mapping attribute clauses determine how an *original* variable is mapped to a *corresponding* variable in a device data environment**

□ original: the variable on the host
□ corresponding: the variable on the device
□ the corresponding variable in the device data environment may share storage with the original variable

Copying x back is not necessary: it was not changed.

Controlling data with the map clause

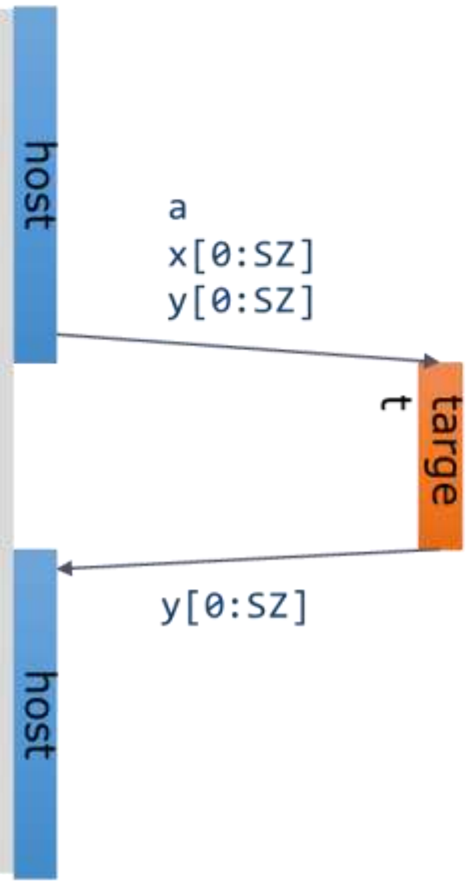
```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement defined from the *host* perspective.

- The various forms of the map clause
  - **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
  - **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables on the host (device to host copy). On entering the region, the initial value of the variables on the device is not initialized.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end).
  - **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to **map(tofrom:list)**.

Example: saxpy

```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:SZ]) \
                   map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



Parallelism on the device

The target construct transfers the control flow to the target device

-- Transfer of control is sequential and synchronous

OpenMP separates offload and parallelism

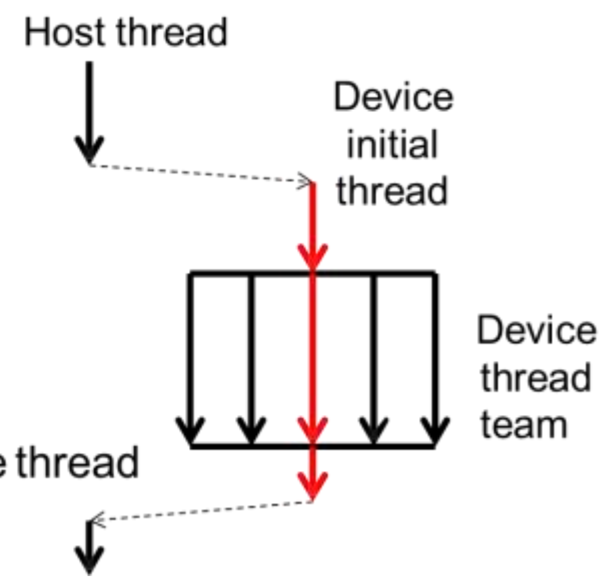
-- Programmers need to explicitly create parallel regions on the target device

-- There are a few useful subset of OpenMP features for a target device such as a GPU

Parallel threads

- Recall fork-join model and parallel regions on a CPU:
  - #pragma omp parallel
- Threads are created on entry to parallel region
- All those threads belong to one **team**
- Threads in a team can synchronize:
  - #pragma omp barrier

```
#pragma omp target
#pragma omp parallel for
for (i=0;i<N;i++)
    ...
```



Transfer control of execution to a **SINGLE** device thread
Only one **team** of threads workshares the loop

Example

```
void saxpy(float a, float* x, float* y,
           int sz) {
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom(y[0:sz])
#pragma omp parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

GPUs are multi-level devices: SIMD, threads, thread blocks

Create a team of threads to execute the loop in parallel using SIMD instructions.

Example, Calculate Pi with target and loop directives

```
#include <omp.h>
#include <stdio.h>
static long numsteps = 100000000;
int main() {
double sum = 0.0;

double step = 1.0 /  double ) num steps ;

#pragma omp target map(tofrom:sum)
#pragma omp loop reduction (+:sum)
for (int i=0; i<numsteps; i++){
     double x = (i + 0.5) * step;
     sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
printf(" pi with %ld steps is %lf\n", num steps, pi);
```

‘teams’ and ‘distribute’ constructs

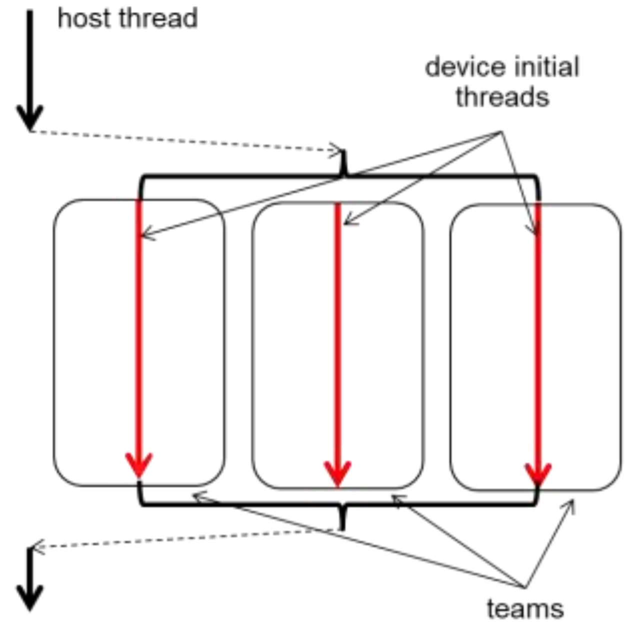
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of *teams*
  - Each team in the league starts with one initial thread – i.e. a team of one thread
  - Threads in different teams **cannot** synchronize with each other
  - The construct must be “perfectly” nested in a **target** construct

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Multiple teams

- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
for (i=0;i<N;i++)
    ...
```



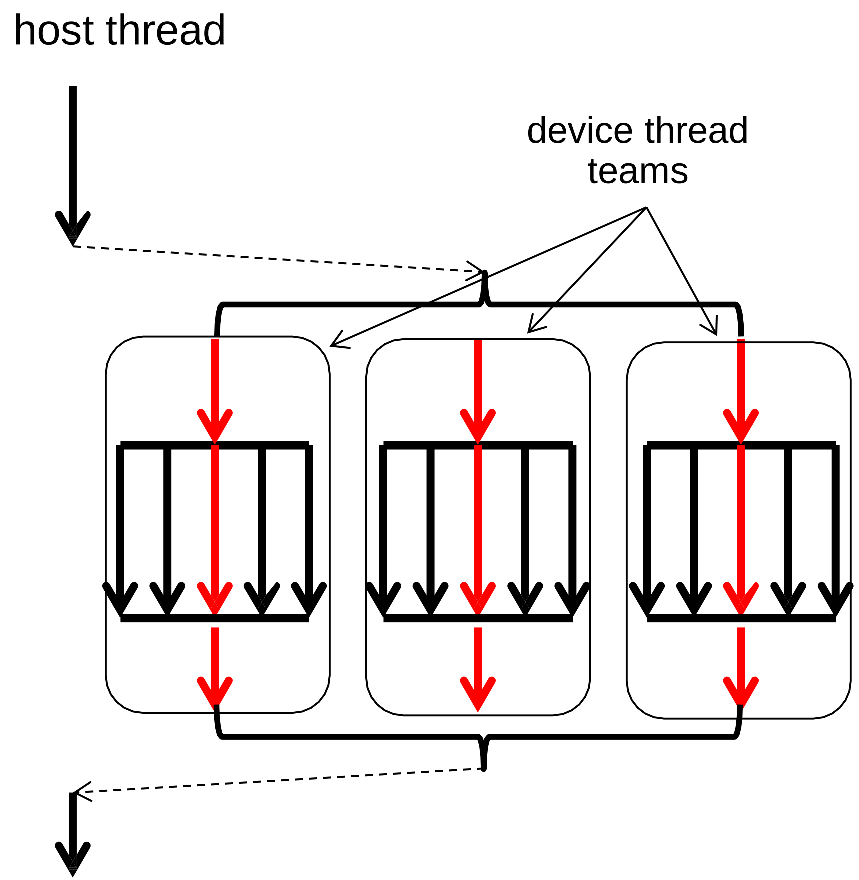
- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Note: number of teams is implementation defined, good for portable performance. Compilers can choose how they map teams and threads.

Multi level parallelism, put it together

- teams distribute
- parallel for simd

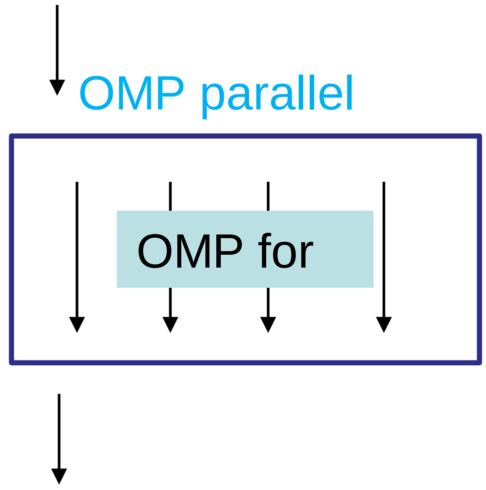
```
#pragma omp target
#pragma omp teams distribute
for (i=0;i<N;i++)
#pragma omp parallel for simd
for (j=0;j<M;j++)
…
```



- Transfer execution control to MULTIPLE device initial threads (one per team)
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the master thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for simd)

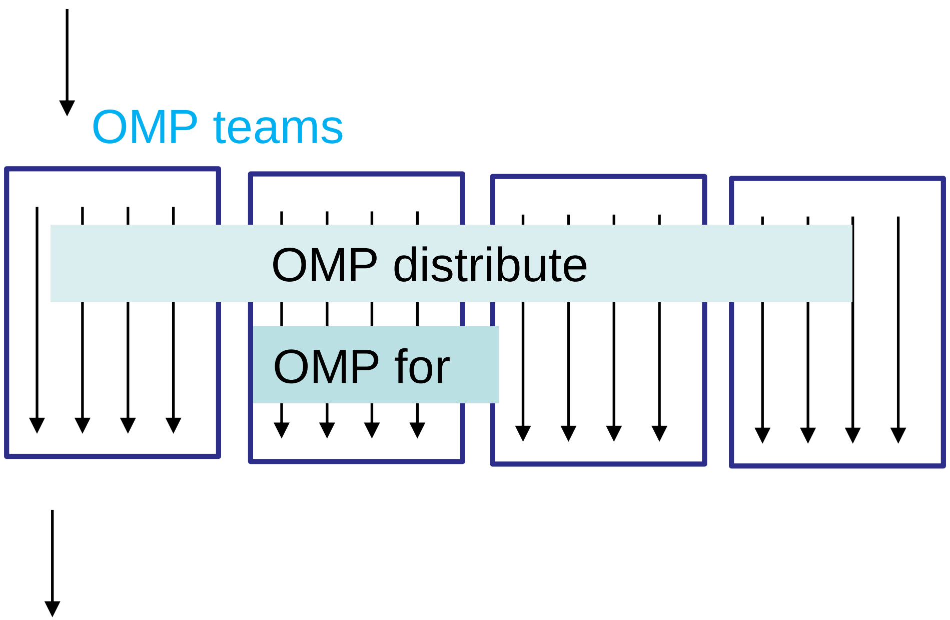
Example (OpenMP code for CPUs)

```
#progma omp parallel for reduction(max:error)
for (int j=1; j < n-1; j++) {
   for (int i=1; i< m-1; i++) {
        newA[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                        + A[j-1][i] + A[j+1][i]);
        error = fmax ( error, fabs (newA[j][i] – A[j][i]));
    }
}
```

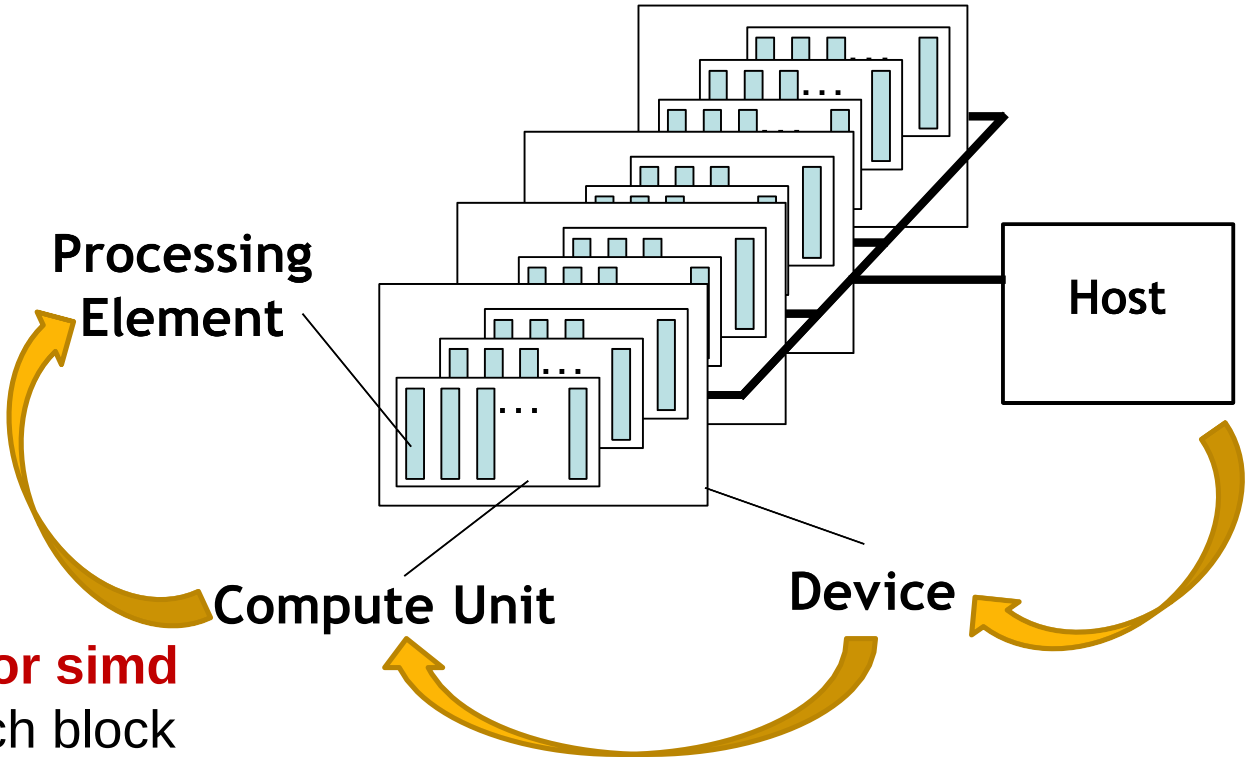


Example (OpenMP code for GPUs)

```
#progma omp target teams distribute reduction (max:error)
for (int j=1; j < n-1; j++) {
#progma omp parallel for reduction(max:error)
   for (int i=1; i< m-1; i++) {
        newA[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                              + A[j-1][i] + A[j+1][i]);
        error = fmax ( error, fabs (newA[j][i] – A[j][i]));
    }
}
```



OpenMP host/device model: Summary



target construct to get onto a device

parallel for simd to run each block of loop iterations on the processing elements

teams construct to create a league of teams with one team of threads on each compute unit

distribute construct to assign blocks of loop iterations to teams

More Directives and Functions for Devices

omp target data: Creates a device data environment and execute the construct on the same device. The target construct specifies that the region is executed by a device and the encountering task waits for the device to complete the target region

omp target enter data

omp target exit data

omp target update: Makes the corresponding list items in the device data environment consistent with their original list items

omp declare target: marks function(s) that can be called on the device

omp get team num()
omp get team size()
omp get num devices()
omp_get_default_device()

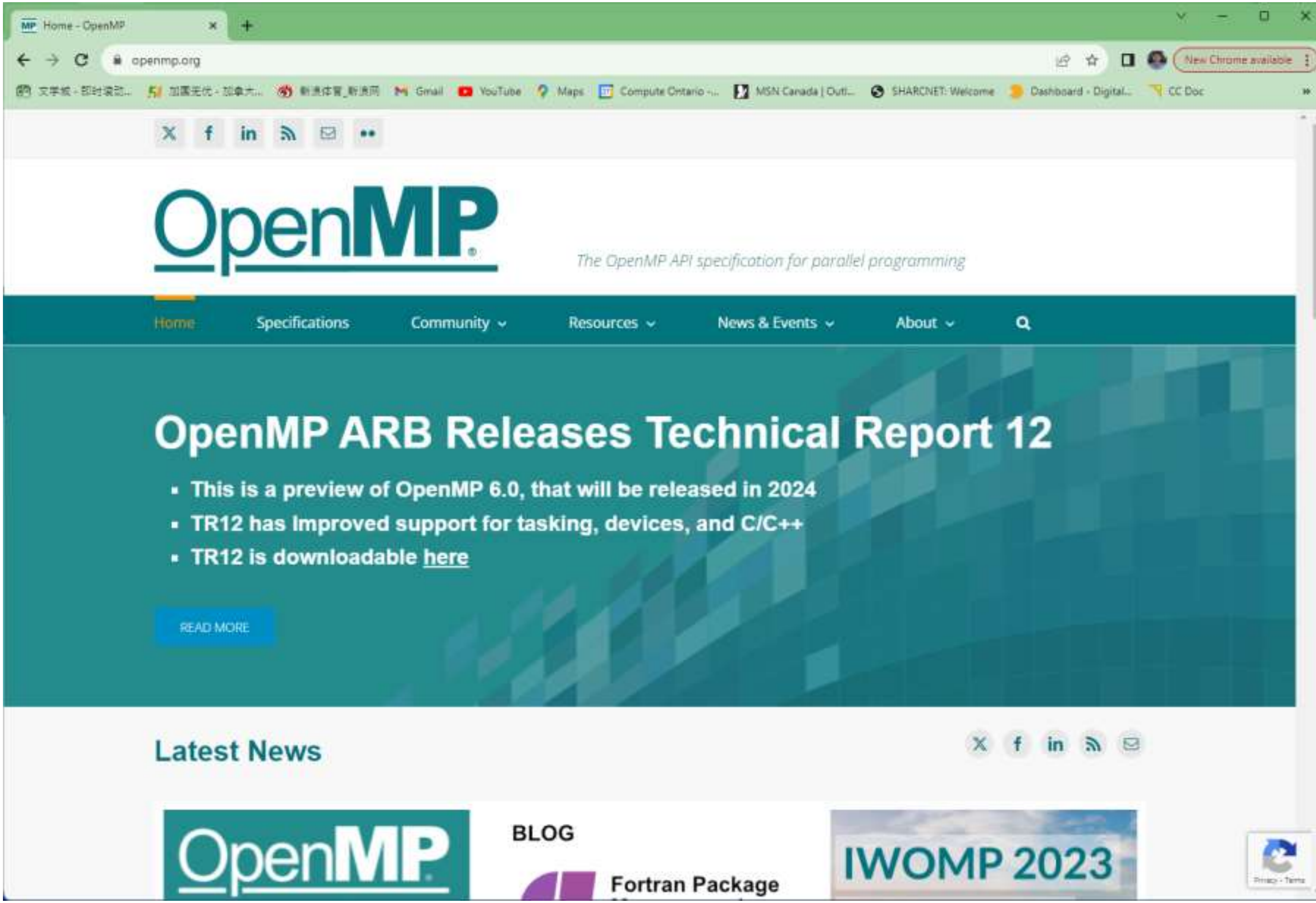
Example

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
{
    #pragma omp target device(0)
    #pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

    #pragma omp target update device(0) to(input[:N])

    #pragma omp target device(0)
    #pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
}
```



Home - OpenMP
openmp.org
Gmail
YouTube
Maps
Compute Ontario -...
MSN Canada | Outl...
SHARCNET: Welcome
Dashboard - Digital...
CC Doc
New Chrome available
OpenMP
The OpenMP API specification for parallel programming
Home
Specifications
Community
Resources
News & Events
About
OpenMP ARB Releases Technical Report 12
This is a preview of OpenMP 6.0, that will be released in 2024
TR12 has Improved support for tasking, devices, and C/C++
TR12 is downloadable here
READ MORE
Latest News
OpenMP
BLOG
Fortran Package
IWOMP 2023

Resources

SC23, Programming Your GPU with OpenMP A “Hands-On” Introduction.

Tom Deakin, Simon McIntosh-Smith, Tim Mattson

https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

https://www.youtube.com/watch?v=uVcvecgdW7g

https://www.youtube.com/watch?v=qEp25Kqjm4o

https://www.youtube.com/watch?v=XI1rRMQnC3g

https://www.youtube.com/watch?v=9w_2tj2uD4M