OpenMP Offloading
-- getting started through examples --

by

Kent Milfeld

milfeld@tacc.utexas.edu

https://tinyurl.com/tacc-2025-appi
https://github.com/milfeld/openmp_offload

Go Parallel

KFM

OpenMP Offloading

Required Proficiency:

- Some experience or basic knowledge of OpenMP
- Working knowledge of C/C++ or F90
- Basic GPU Architecture Understanding

References: **openmp.org**/specifications

6.0 Specification
6.0 Examples

Basic operations/concepts of code are explained in tutorial.

(Non-tutorial viewers should read grammar/syntax text after code sections first, then peruse code.)

Agenda

We will not cover Unified Shared Memory.

🟪OpenMP

- Background and Levels of Parallelism.
- GPU Programming.
- Review Worksharing Loop.
- Quick Intro to Simple Tasking. ←

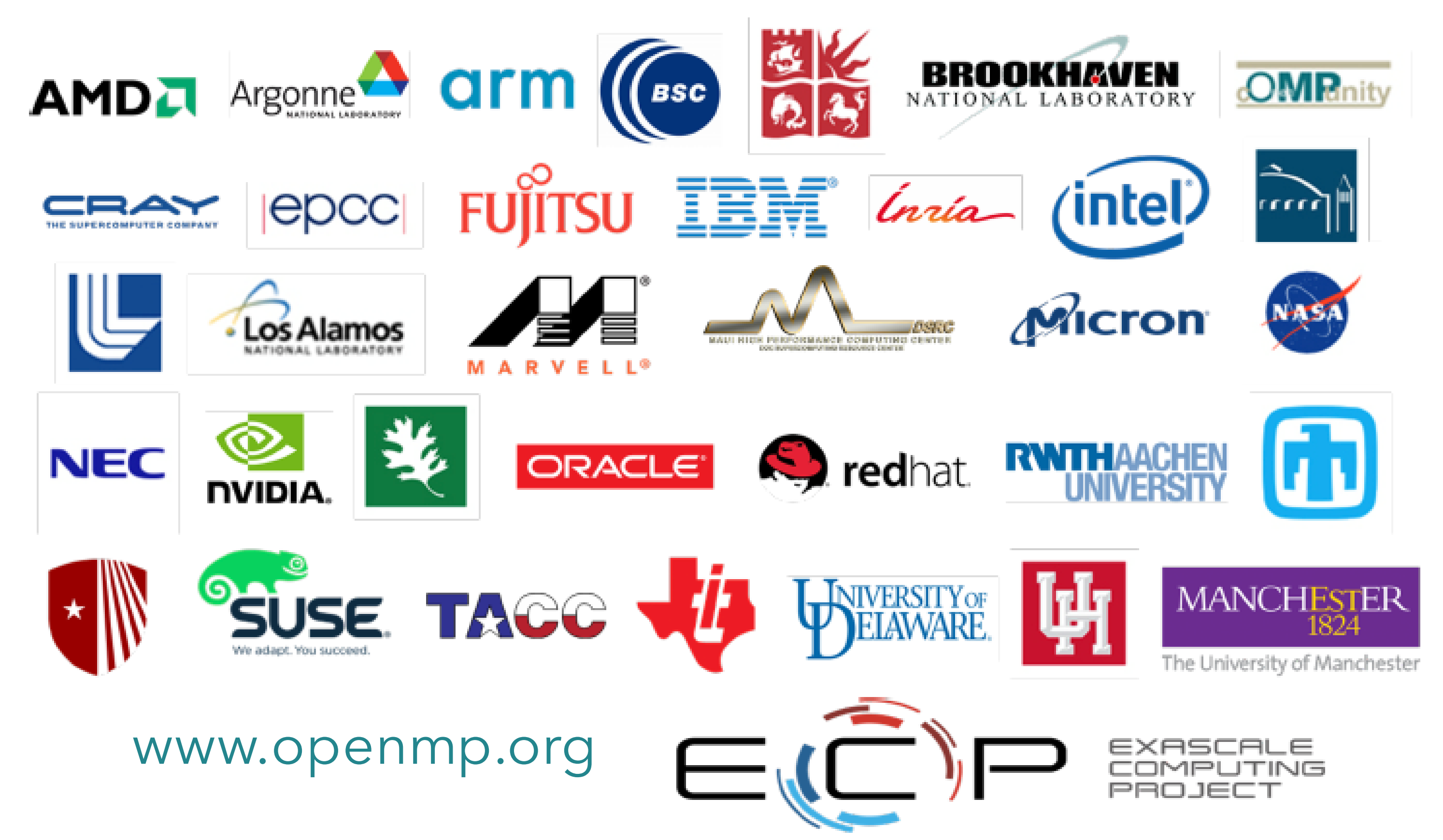
Useful for understanding modern OpenMP, but not necessary for basic offloading.

OpenMP Offloading

- Launch and direct device execution with target.
- Explicit/Implicit mapping ("transfer") to/from device.
- teams, distribute, and loop.
- Data Persistence: target data.
  Data Movers: target enter/exit data, target update.
- declare target (ensure function/data execution/access on device).
  Asynchronous Offloading with nowait and depend.

OpenMP -- board and membership

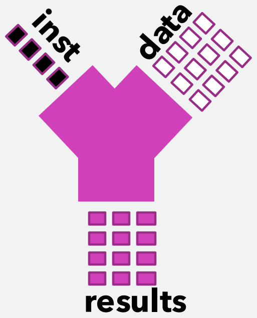
Since 1997
A non-profit organization
Governed by OpenMP Architecture Review Board (ARB)


AMD
Argonne NATIONAL LABORATORY
arm
BSC
BROOKHAVEN NATIONAL LABORATORY
OMPunity
CRAY THE SUPERCOMPUTER COMPANY
epcc
FUJITSU
IBM
Inria
intel
Los Alamos NATIONAL LABORATORY
MARVELL
MAUI HIGH PERFORMANCE COMPUTING CENTER
Micron
NASA
NEC
NVIDIA
ORACLE
redhat
RWTH AACHEN UNIVERSITY
SUSE We adapt. You succeed.
TACC
UNIVERSITY OF DELAWARE
UH
MANCHESTER 1824 The University of Manchester
www.openmp.org
ECP EXASCALE COMPUTING PROJECT

OpenMP deals with multiple levels of Parallelism

parallel do/for, task — Across Cores

simd — Across SIMD Lanes



target — Across Devices

Modern OpenMP -- beyond Worksharing Loops

Just a few directives and clauses.

Version	Directives	Parallelism
pre-3.0	parallel for|do	worksharing loops
3.0	task	explicit tasking, dependence scheduling
4.0	simd	SIMD
	target	device offloading
	(teams distribute, loop and *data-mapping)*	
:		
6.0	...	

Other important directives and clauses.

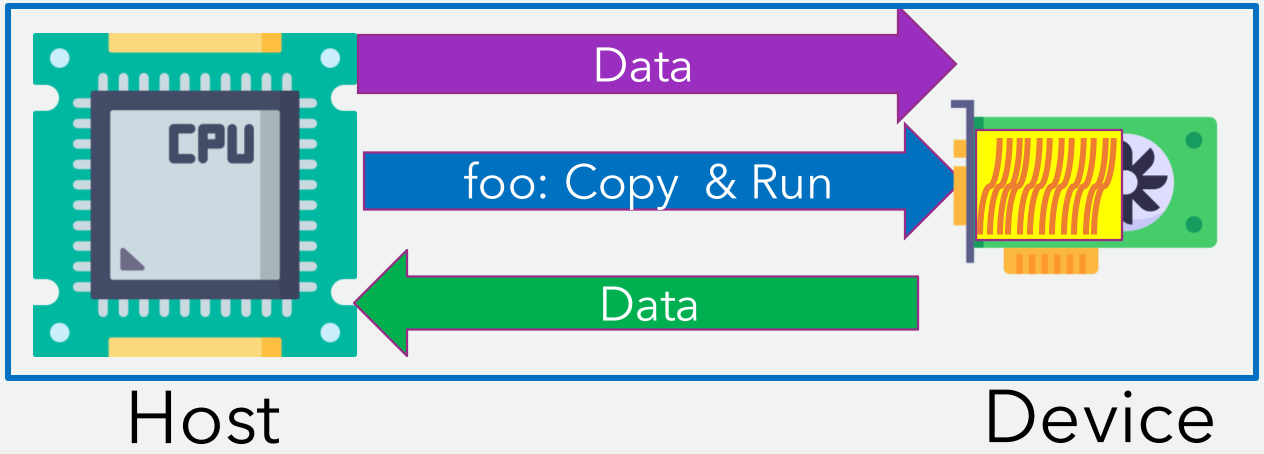
CPU affinity	(5.0)
loop transforms	(5.1-6.0)
free-agent threads	(6.0)
taskgraphs	(6.0)

Computing on GPU (with separate host and device storage)

```
compile prog.c   # creates fat binary
./a.out          # run on host, exec foo binary on gpu
```

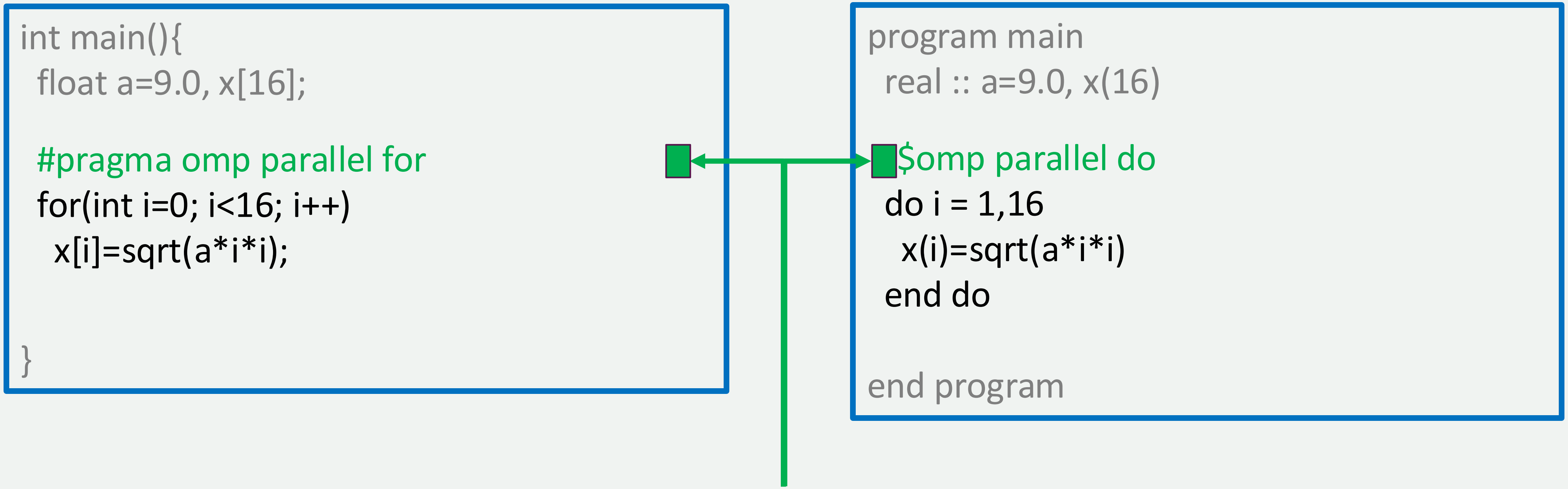
h_ == host   d_ == device

```
declare foo a gpu cuda kernel

main() {
declare h_x,    d_x (pointer)
…
    alloc_on_dev( sz            d_x )
    copy_to_dev(       h_x -> d_x )
    foo(d_x)
    copy_from_dev( h_x <- d_x )
    dealloc_on dev(             d_x )
…
```



CUDA programs looks like this, where it all began.

Worksharing Loop



```
int main(){
  float a=9.0, x[16];

  #pragma omp parallel for
  for(int i=0; i<16; i++)
    x[i]=sqrt(a*i*i);


}
```

```
program main
  real :: a=9.0, x(16)

  $omp parallel do
  do i = 1,16
    x(i)=sqrt(a*i*i)
  end do

end program
```

creates *team* of threads (e.g. export OMP_NUM_THREADS=4)

implicit tasks (iteration chunks of work) executed by threads

Worksharing Loop

```
void foo(float a, float *x, int n){

printf("id=%d\n",omp_get_thread_num());

  #pragma omp for
  for(int i=0; i<n; i++)
    x[i]=sqrt(a*i*i);
}

int main(){
  float a=9.0, x[16];
  #pragma omp parallel
  {
     foo(a,x,16);
  }
}
```

```
subroutine foo(a, x, n)
  use omp_lib
  real :: a, x(*)

  print*, "id=",omp_get_thread_num()

  !$omp do
  do i = 1, n
    x(i)=sqrt(a*i*i)
  end do
end subroutine

program main
  real :: a=9.0, x(16)
  !$omp parallel
    call foo(a,x,16)
  !$omp end parallel
end program
```

parallel creates *team* of threads

for / do workshares implicit tasks among threads

OpenMP Tasking with Dependences

Makes a bundle of work (task) for asynchronous execution.

- **task** construct -- creates a task out of a code block or function to be queued for execution by any thread of team.

- **depend** clause- orders tasks
  - ❑ between tasks identified by variable
  - ❑ ordering determined by dependence type

task

C/C++

```
   void init(float *x)          { for(int i=0; i<N; i++) x[i] = i;  }
   void axpy(float *x, float *y){ for(int i=0; i<N; i++) y[i] = 2*x[i] + y[i]; }
int main(){
  float x[N], y[N];


  {

    init(y);      // work (task) on y   -   write to y    (out: y)


    init(x);      // work (task) on x   -   write to x    (out: x)


    axpy(x,y);    // work (task) on x,y -   read x        (   in: x)
              //                      read/write y   (inout: y)



    for(int i=0; i<N; i++){ x[i]=sqrt(y[i]); }  // special tasking for loops
  }
}
```

OpenMP "speak"

task

C/C++

```
    void init(float *x)          { for(int i=0; i<N; i++) x[i] = i;  }
    void axpy(float *x, float *y){ for(int i=0; i<N; i++) y[i] = 2*x[i] + y[i]; }
int main(){
  float x[N], y[N];

  #pragma omp parallel master  // Get team of threads; 1 thread creates tasks.
  {
    #pragma omp task          // Race here. Any team thread can execute concurrently.
    init(y);

    #pragma omp task          // Race here. Any team thread can execute concurrently.
    init(x);

    #pragma omp task          // Race here. Any team thread can execute concurrently.
    axpy(x,y);



    #pragma omp taskloop  // Race here. taskloop chunks iterations into tasks.
    for(int i=0; i<N; i++){ x[i]=sqrt(y[i]); } // Any team thread can execute chunks.
  }
}
```

task with depend clause

C/C++

```
   void init(float *x)         { for(int i=0; i<N; i++) x[i] = i;  }
   void axpy(float *x, float *y){ for(int i=0; i<N; i++) y[i] = 2*x[i] + y[i]; }
int main(){
  float x[N], y[N];

  #pragma omp parallel master
  {
    #pragma omp task depend(out: y)     // Task1 writes to y
    init(y);

    #pragma omp task depend(out: x)     // Task2 writes to x
    init(x);

    #pragma omp task depend(in: x) depend(inout: y) // Task3 reads x, reads/writes y
    axpy(x,y);                               // Task3 runs after Task1 & Task2

    #pragma omp taskwait                // Wait for all (sibling) tasks to complete.

    #pragma omp taskloop                // could use depend here and avoid taskwait
    for(int i=0; i<N; i++){ x[i]=sqrt(y[i]); } // execute iteration chunks in parallel
  }
}
```

Task 1 & 2 can run concurrently

task

```
subroutine init(x,n)
real :: x(n)
  do i=1,n; x(i) = i; enddo
end subroutine
subroutine axpy(x,y,n)
real :: x(n),y(n)
  do i=1,n; y(i)=2*x(i)+ y(i); enddo
end subroutine
```

Fortran

OpenMP "speak"

```
program main
  integer, parameter ::   N=16
  real              :: x(N), y(N)


   call init(x,N)    !! work (task) on x -   write to x   (out: x)


   call init(y,N)    !! work (task) on y -   write to y   (out: y)


   call axpy(x,y,N)  !! work (task) on x,y - read from  x (   in: x)
                 !!                  read/write y (inout: y)


   do i=1,N; x(i)=sqrt(y(i)); enddo  !! special tasking for loops

end program
```

task

Fortran

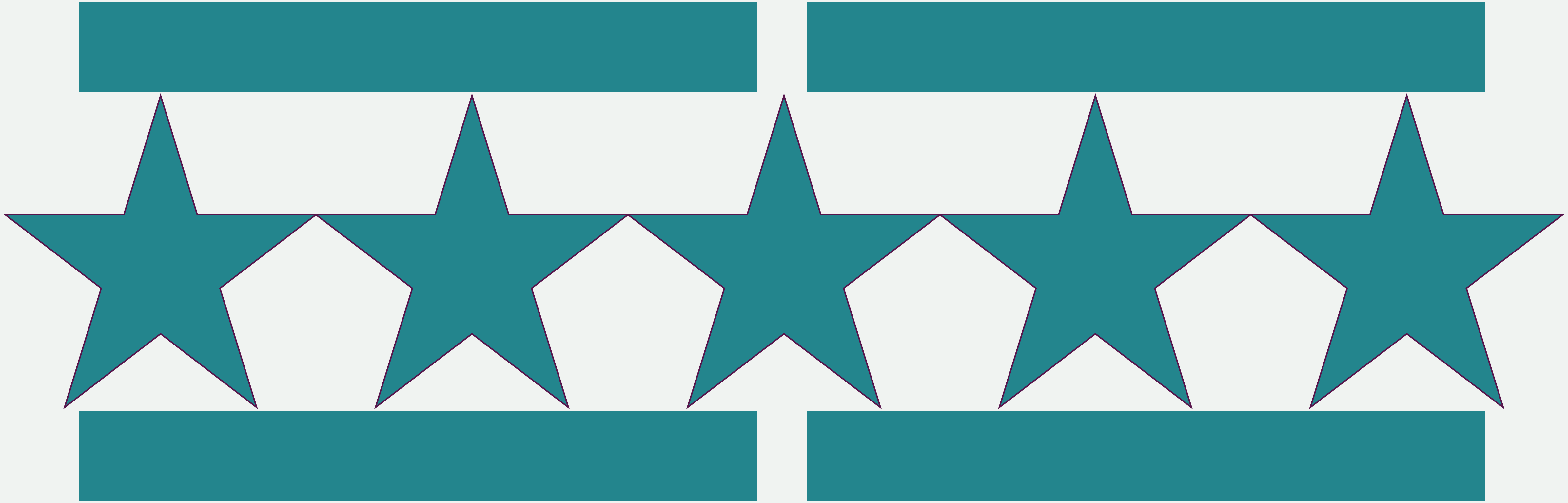
```
program main
  integer, parameter ::   N=16
  real              :: x(N), y(N)

   !$omp parallel master      !! Get team of threads; 1 thread creates tasks
   !$omp task              !! Race here. Any team thread can execute concurrently.
   call init(x,N)
   !$omp end task

   !$omp task              !! Race here. Any team thread can execute concurrently.
   call init(y,N)
   !$omp end task

   !$omp task              !! Race here. Any team thread can execute concurrently.
   call axpy(x,y,N)
   !$omp end task



   !$omp taskloop             !! Race here. taskloop chunks iterations into tasks.
   do i=1,N; x(i)=sqrt(y(i)); enddo     !! Any team thread can execute chunks.
  !$omp end parallel master

end program
```

task with depend clause

Fortran

```
program main
  integer, parameter ::   N=16
  real              :: x(N), y(N)

 !$omp parallel master
  !$omp task depend(out: y)              !! Task1 writes to y.
  call init(x,N)
  !$omp end task

  !$omp task depend(out: x)              !! Task2. writes to x.
  call init(y,N)
  !$omp end task

  !$omp task depend(in: x) depend(inout: y) !! Task3 reads x, reads/writes y.
  call axpy(x,y,N)                      !! Task3 runs after Task1 & Task2.
  !$omp end task

  !$omp taskwait                        !! Wait for all (sibling) tasks to complete.

  !$omp taskloop                        !! could use depend here and avoid taskwait.
  do i=1,N; x(i)=sqrt(y(i)); enddo      !! execute iteration chunks in parallel.

 !$omp end parallel master

end program
```

Task 1 & 2 can run concurrently



OPENMP OFFLOADING

Agenda

OpenMP Offloading

- ■ Launch and direct device execution with target.
- Explicit/Implicit mapping (“transfer”) to/from device.
- teams, distribute, and loop.
- Data Persistence: target data.

  Data Movers: target enter/exit data, target update.
- declare target (ensure function/data execution/access on device).

  Asynchronous Offloading with nowait and depend.

target (offloading)

- #pragma omp target

- !$omp target

C/C++

```
int main(){
 float a=9.0, x[16];

 #pragma omp target
    foo(a,x);


 #pragma omp target
 {
  for(int i=0; i<16; i++)
    x[i]=sqrt(a*i*i);
 }

}
```

Fortran

```
program main
 real :: a=9.0, x(16)

 !$omp target
  call foo(a,x)
 !$omp end target

 !$omp target
  do i = 1,16
   x(i)=sqrt(a*i*i)
  end do
 !$omp end target

end program
```

Runs on device with only 1 thread.

Implicit data management: alloc device storage & copy data to device; dealloc & copy from at end of target region.

target clauses – for beginners

```
#pragma omp target  [ clause [clause …] ]     C/C++
          !$omp target  [ clause [clause …] ]     Fortran
```

device(dev_id)	*device #, default 0*
if(int expr)	*if false fallback to host (“! 0” in C/C++, logical in F90)*
map(map-type: var) (Explained in Next Section.)	*alloc/dealloc storage for var on device & copy data between host and device.* *on host var is called original variable/storage* *on device var is called corresponding variable/storage* *map-type keywords: to, from, or tofrom specify copy direction and implicitly handle storage*
nowait, depend	*makes offload async with dependences (with target task)*

Agenda

OpenMP Offloading

- Launch and direct device execution with target.

🟪 Explicit/Implicit mapping (“transfer”) to/from device.

- teams, distribute, and loop.
- Data Persistence: target data.

  Data Movers: target enter/exit data, target update.
- declare target (ensure function/data execution/access on device).

  Asynchronous Offloading with nowait and depend.

Explicit Data Mapping

C/C++

```
#define N 1048576
int main(){
 int scal, array[N], ptee[N], *ptr;

 scal = 0;
 for(int i=0; i<N; i++){ array[i] = 0; ptee[i] = 0; } // inits
 ptr=&ptee[0];

// Explicit Data Mapping (Defaults to tofrom map-type)

 #pragma omp target map(scal, array, ptr[0:N])
 {
   scal = scal + 1;
   for(int i=0; i<N; i++){ array[i] = array[i] + scal; }
   for(int i=0; i<N; i++){   ptr[i] =   ptr[i] + scal; }
 }
}
```

alloc storage
cp data **to** dev

cp data **from** dev
dealloc storage

Array Section [lb:len:stride] specifies storage pointed to by the pointer.

Efficient Explicit Data Mapping with map-types

C/C++

```
#define N 1048576
int main(){
  int scal, array[N], ptee[N], *ptr;

  scal = 1;
  for(int i=0; i<N; i++){ array[i] = 0; ptee[i] = 0; } // inits
  ptr=&ptee[0];

// Explicit Data Map(copy scal to, and array and ptee from)

#pragma omp target map(to: scal) map(from: array, ptr[0:N])
  {

     for(int i=0; i<N; i++){ array[i] = i + scal; }
     for(int i=0; i<N; i++){   ptr[i] = i + scal; }
  }
}
```

Implicit behavior – first private and data mapping

C/C++

```
#define N 1048576
int main(){
  int scal, array[N], ptee[N], *ptr;

  scal = 0;
  for(int i=0; i<N; i++){ array[i] = 0; ptee[i] = 0; } // inits
  ptr=&ptee[0];

// Implicit Behavior: Works as expected except scalars are firstprivate

  #pragma omp target:
  {
    scal = scal + 1;
    for(int i=0; i<N; i++){ array[i] = array[i] + scal; }
    for(int i=0; i<N; i++){   ptr[i] =   ptr[i] + scal; }
  }
}
```

scal **not** copied to host

Explicit Data Mapping

Fortran

```
program main
 integer, parameter :: N=1048576
 integer            :: scal, array(N)
 integer,pointer    :: ptr(:)
 integer,target     :: ptee(N)

 scal = 0
 do i=1,N;  array(i) = 0; ptee(i) = 0; enddo   !! inits
 ptr=>ptee

 !! Explicit Data Mapping (Defaults to tofrom map-type)

 !$omp target map(scal) map(array) map(ptr)
   scal = scal + 1;
   do i=1,N;  array(i) = array(i)+1; enddo
   do i=1,N;  ptr(i)   =   ptr(i)+1; enddo
 !$omp end target

end program
```

alloc storage
cp data **to** dev

cp data **from** dev
dealloc storage

Efficient Explicit Data Mapping with map-types

Fortran

```
program main
 integer, parameter :: N=1048576
 integer            :: scal, array(N)
 integer,pointer    :: ptr(:)
 integer,target     :: ptee(N)

 scal = 0
 do i=1,N;  array(i) = 0; ptee(i) = 0; enddo   !! inits
 ptr=>ptee

 // Explicit Data Map(copy scal to, and array and ptr from)

 !$ omp target map(to: scal) map(from: array, ptr)


   do i=1,N;  array(i) = array(i) + scal; enddo
   do i=1,N;  ptr(i)   =   ptr(i) + scal; enddo
 !$omp end target

end program
```

Implicit behavior – first private and data mapping

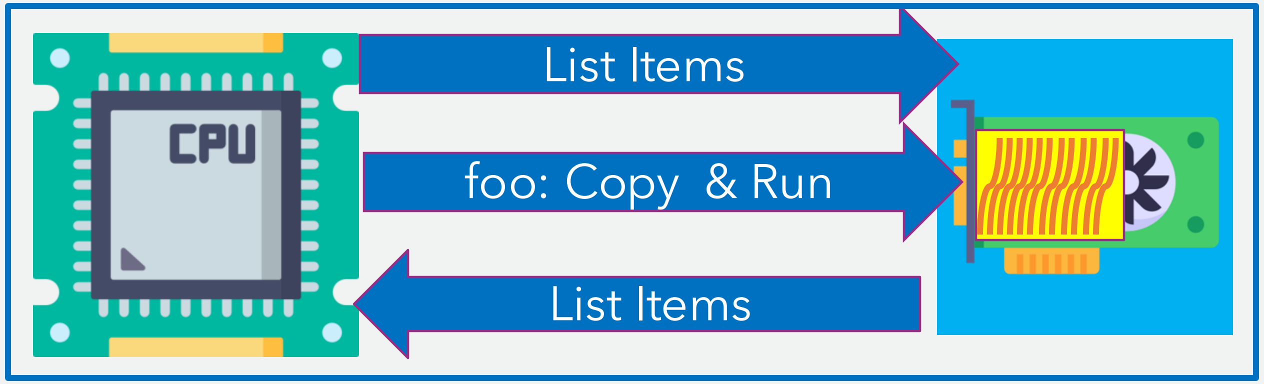
Fortran

```
program main
  integer, parameter :: N=1048576
  integer            :: scal, array(N)
  integer,pointer    :: ptr(:)
  integer,target     :: ptee(N)

  scal = 0
  do i=1,N;  array(i) = 0; ptee(i) = 0; enddo   !! inits
  ptr=>ptee

  ! Implicit Behavior: Works as expected except scalars are firstprivate

  !$omp target
     scal = scal + 1;
     do i=1,N;  array(i) = array(i)+1; enddo
     do i=1,N;  ptr(i)   =   ptr(i)+1; enddo
  !$omp end target

end program
```

scal value not copied to host

map clause

What did we just do with map?



Host

host data environment
original variables

Device

device data environment
corresponding variables

map(***action:*** ***variables***)

default ***action*** (map-type) is to and from (tofrom) device

variables in lexical extent of target region are implicitly mapped

map clause syntax

map(*modifier:* *list items*)

modifiers

map-types

to	allocate & cp to
from	cp from & deallocate
tofrom	cp to & from
alloc	allocate only (**storage** in 6.0)

always

always	always perform the map-type

list items ("variables")

OMP	**C/C++**	**F90**
---------------	-----------	-------------------------
scalars	int a;	integer ::a
arrays	int a[8];	integer :: a[N]
pointers	int *ptr;	integer,pointer::a
objects (structures)	struct ...	data type ..
array sections	a[*start:extent,stride*] (used only in OMP directives)	

Other modifiers: delete, present, close, self, ref, mapper, iterators.

Agenda

OpenMP Offloading

- Launch and direct device execution with target.
- Explicit/Implicit mapping (“transfer”) to/from device.

■ teams, distribute, and loop.

- Data Persistence: target data.
  Data Movers: target enter/exit data, target update.
- declare target (ensure function/data execution/access on device).
  Asynchronous Offloading with nowait and depend.

Parallel **Offloading** – with teams, distribute, and loop

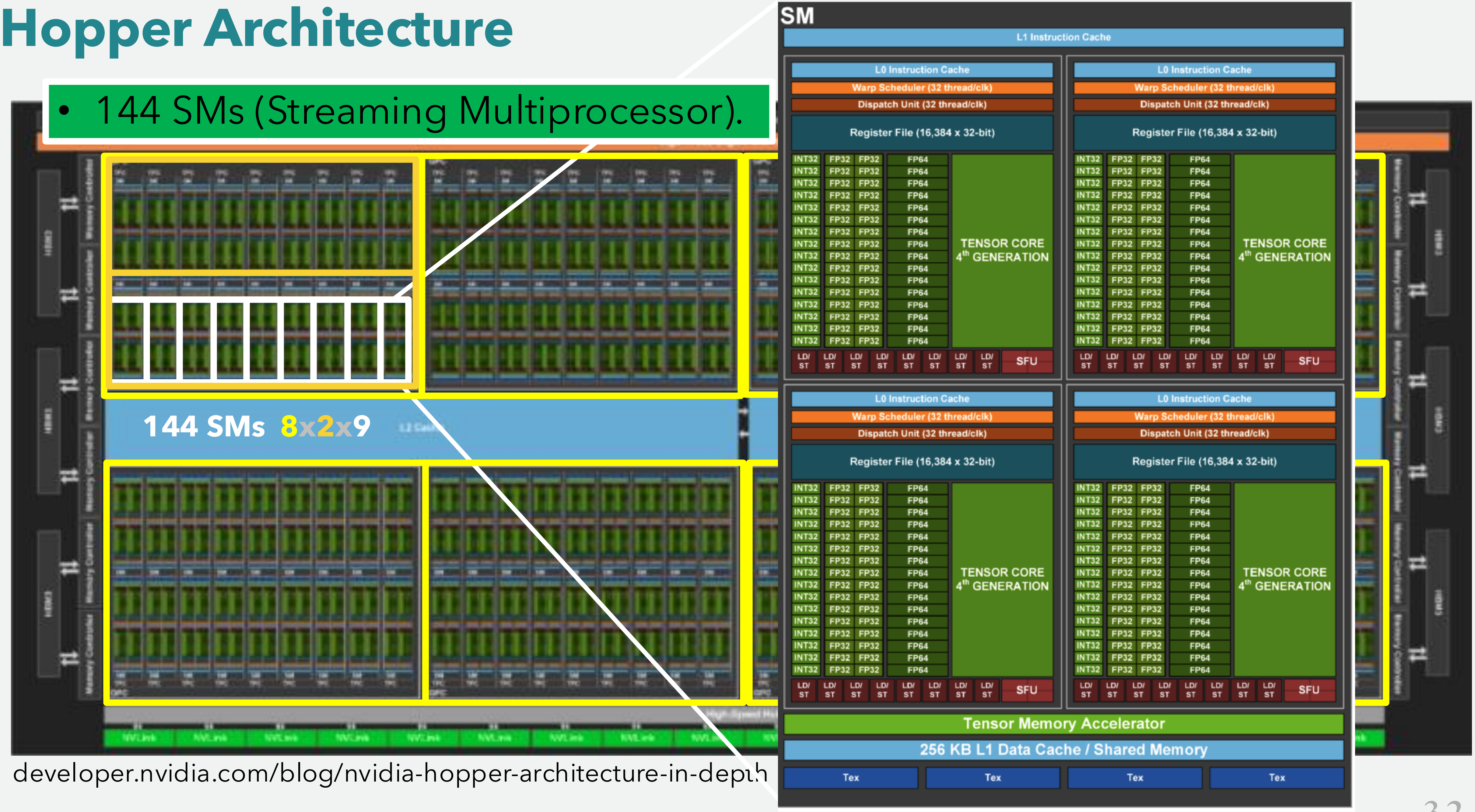
CUDA blocks and threads --> OMP teams and threads

First, GPU architecture:

- ~100 SMs (Streaming Multiprocessors)
- Each SM
  - Is a set of cores (128 FP32, 64 FP64, …, 4 Tensor)
  - Memory is *shared* within set, but not with other SMs
  - Instruction streams (independent instructions) are scheduled on 32 threads (a warp) to execute simultaneously.

Hopper Architecture

- Scheduler manages warps on 32 threads



SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex
NVLink

- 144 SMs (Streaming Multiprocessor).

144 SMs 8x2x9

developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth

teams and parallel with target

Creates teams for the SMs
and threads within an SM

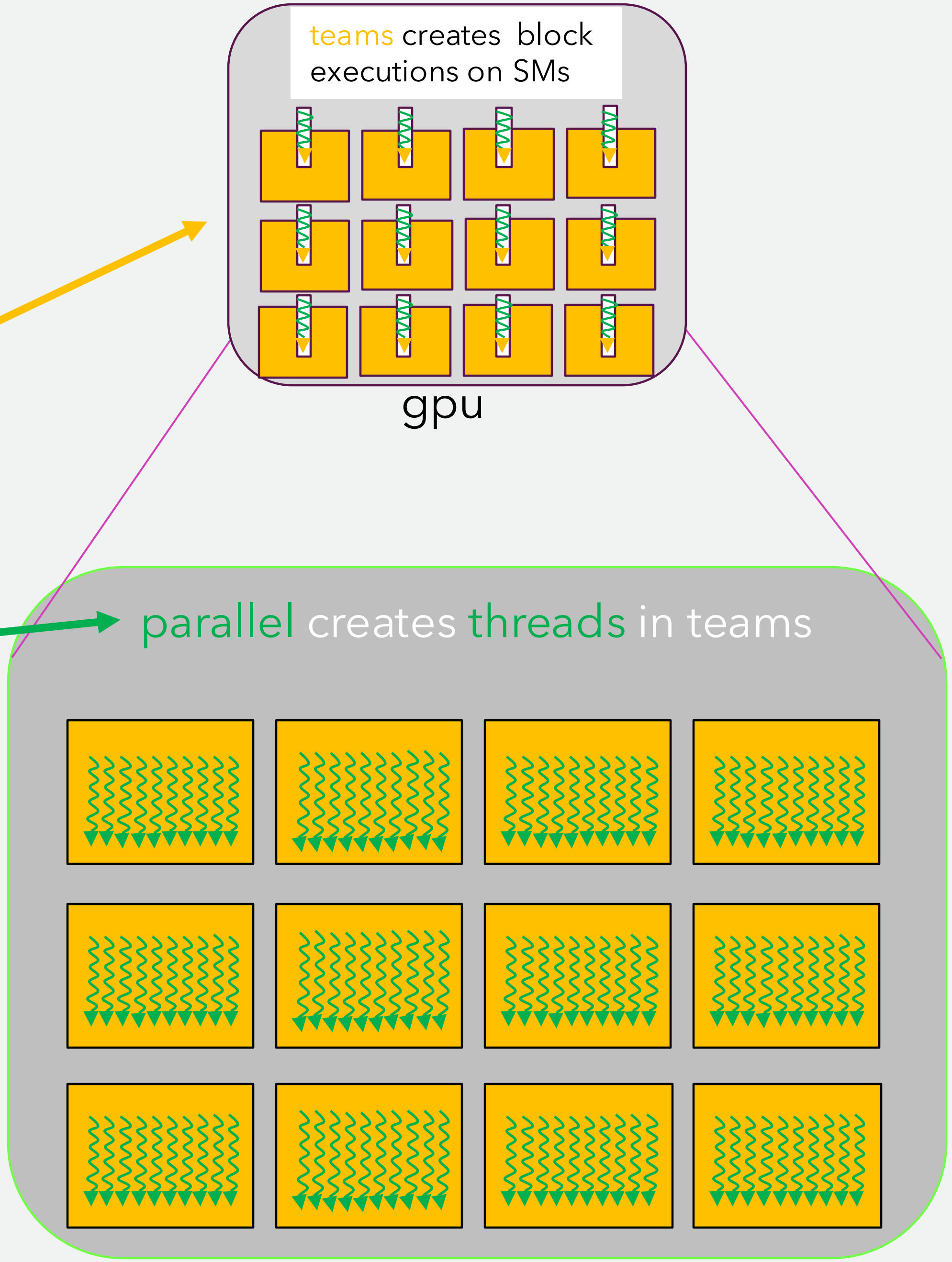
- **teams**: creates teams for execution on SMs
- **parallel**: creates threads within the team
- Default # of threads is implementation defined.

```
#pragma omp target teams
#pragma omp parallel
{
  // code block
}
```

or

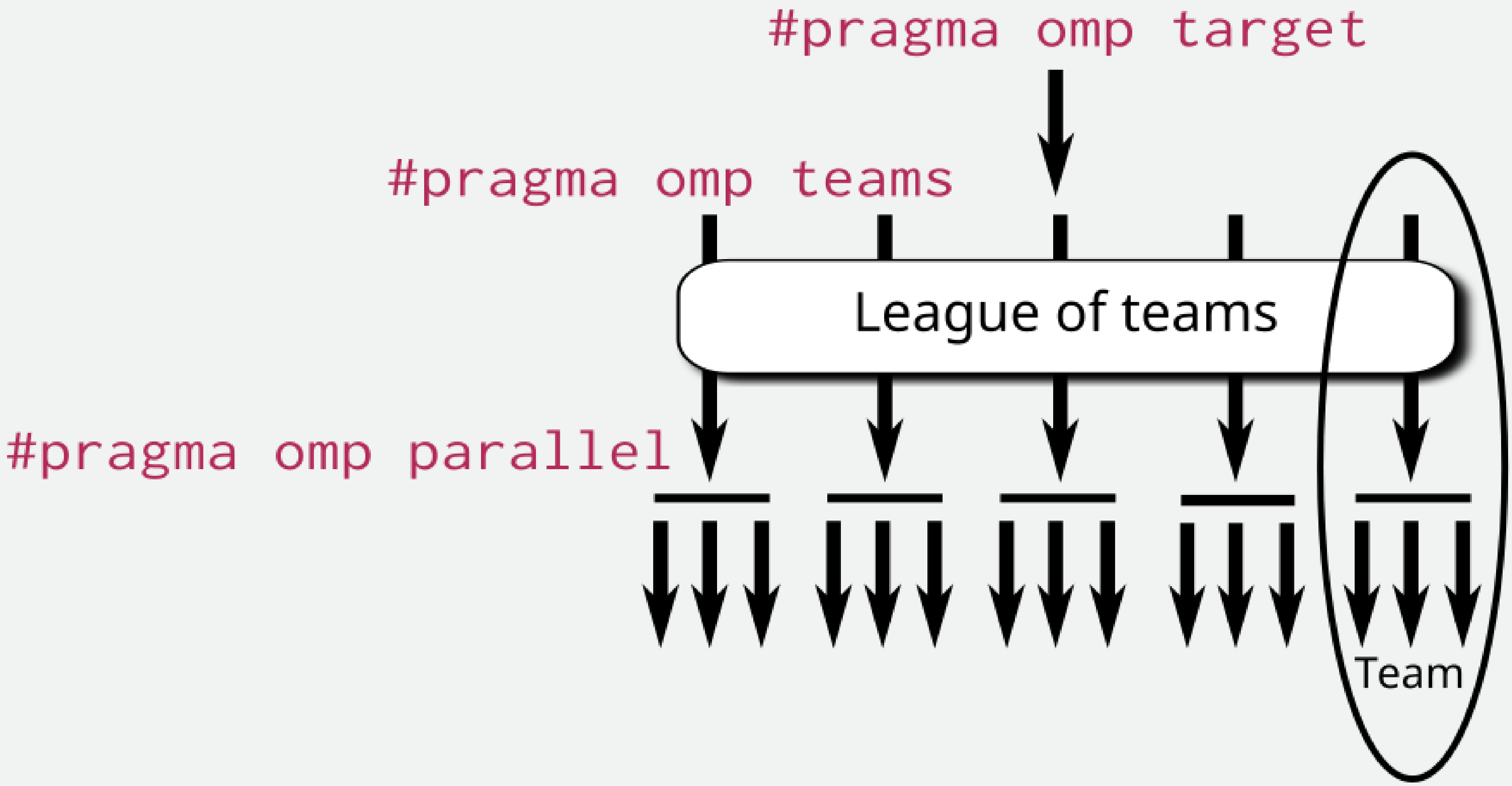
```
#pragma omp target teams parallel
{
  // code block
}
```

all executing the same code block


teams creates block executions on SMs
gpu
parallel creates threads in teams

Without num_threads(), number of threads is implementation defined.

League of multi-threaded teams



work-distribution directives:

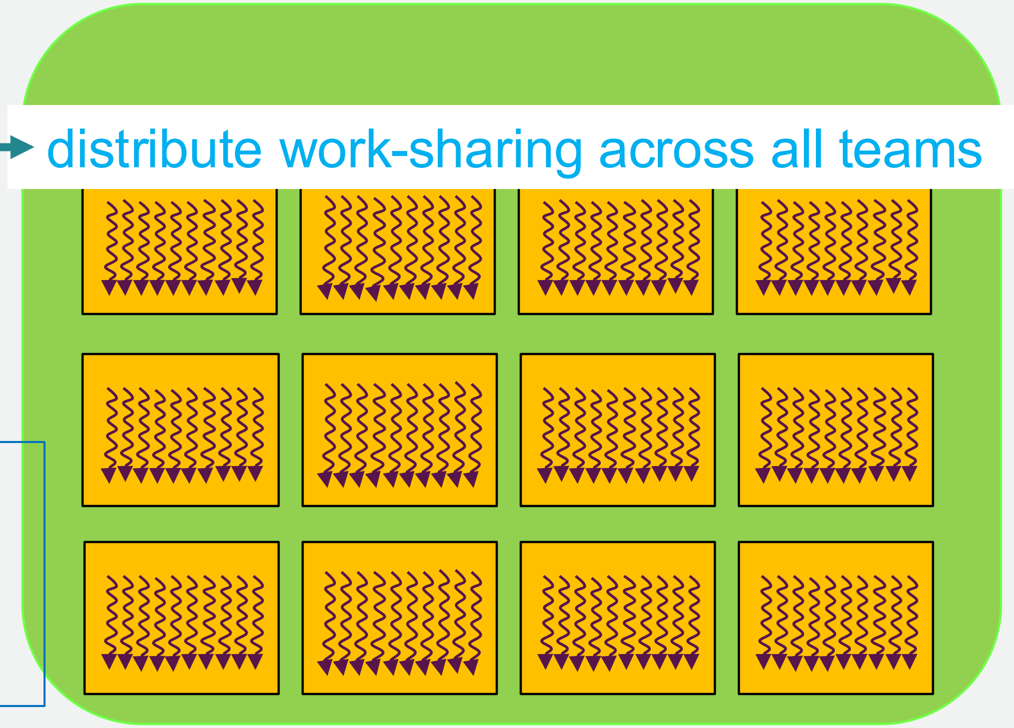
partitioning loop iterations across **teams** and **parallel** constructs

- **distribute** construct specifies iterations of an associated loop are to be distributed across the teams.
- **for** construct is for worksharing across the threads.

```
#pragma omp target teams distribute
#pragma omp parallel for
for (...){
  // code block
}
```

or

```
#pragma omp target teams distribute parallel for
for (...){
  // code block
}
```



distribute work-sharing across all teams

Distribute worksharing with target **teams, distribute parallel for**, and (“new”) **loop**

```
#define N (1 << 27)
int main()
{
  float a=2.0, y[N], x[N];
  for( int i=0; i<N; i++){x[i]=(float)i/N; y[i]=(float)i/N;}



    #pragma omp target teams distribute parallel for
    for (int i=0; i<N; i++) {y[i]=a*x[i]+y[i];}

    #pragma omp target teams loop
    for (int i=0; i<N; i++) {y[i]=a*x[i]+y[i];}

}
```

More control with clauses.

More control by Implementation.

Number of teams and threads are Implementation defined.

Distribute worksharing with **target teams, distribute parallel for**, and (“new”) **loop**

```
program main
  integer, parameter :: N=1024*1024*128
  real  ::  a=2.0, y(N), x(N), rN=N
  do i=1,N; y(i)=i/rN; x(i)=i/rN; end do


    !$omp target teams distribute parallel do
       do i=1,N; y(i)=a*x(i)+y(i); end do


    !$omp target teams loop
       do i=1,N; y(i)=a*x(i)+y(i); end do



end program
```

More control with clauses.

More control by Implementation.

Number of teams and threads are Implementation defined.

loop

Allows compiler to control work distribution and worksharing with loop construct.

- CONS: limited user control (clauses)
- PROS: simple syntax, compiler handles the parallelism

```
#pragma omp target teams
#pragma omp loop
for (...){
  // code block
}
```

or

```
#pragma omp target teams loop
for (...){
  // code block
}
```

```
!$omp target teams
!$omp loop
do i=...
  !! code block
end do
```

or

```
!$omp target teams loop
do i=1...
  !! code block
end do
```

* General: can be use in lieu of parallel for|do on host.

Agenda

OpenMP Offloading

- Launch and direct device execution with target.
- Explicit/Implicit mapping (“transfer”) to/from device.
- teams, distribute, and loop.

■ Data Persistence: target data.

Data Movers: target enter/exit data, target update.

- declare target (ensure function/data execution/access on device).

Asynchronous Offloading with nowait and depend.

target data – Creating a Data-Persistence Region

C/C++

```
int main () {
  int A[2]={10,11};

  #pragma omp target data map(tofrom: A) // map applies to the region {}
  {                              // Here: A is allocated & copied to device

      #pragma omp target    //    (ref count to keep track)
      {
         A[0]++; A[1]++;
      }

      #pragma omp target    //    (ref count to keep track)
      {
         A[0]++; A[1]++;
      }
                       // <= Here: On Host A=10,11; On device A=12,13.

  }                    //<= Here: A is copied from device & deallocated, and
}                     //         Host and Device have same values, A=12,13
```

target data – Creating a Data-Persistence Region

FORTRAN

```
program main
 integer :: A(2)=(/10,11/)

!$omp target data map(tofrom: A) !! map applies to region (here to end target data)
                  !! Here: A is allocated & copied to device


          !$omp target No      !! (ref counter keeping track) implicit mapping
           A(1)=A(1)+1; A(2)=A(2)+1
          !$omp end target



          !$omp target No      !! (ref counter keeping track) implicit mapping
           A(1)=A(1)+1; A(2)=A(2)+1
          !$omp end target

              !! <= Here: On Host A=10,11; On device A=12,13.

!$omp end target data   !!<= Here: A is copied from device & deallocated, and
              !!     Host and Device have same values, A=12,13
end program
```

target data Syntax

- map applies to target data region
  employs reference counter

target data map(*modifiers*: *vars*)

map modifiers

map-type:
to, from, tofrom, alloc
other modifiers: …

clauses

device -- device number(s)
if -- on host | device
nowait -- async (6.0)
depend -- task ordering (6.0)
… -- many others (6.0)

target enter/exit data -- simple example

```
#include <iostream>
using namespace std;

int main(){
  int A[2]={10,11};

  #pragma omp target enter data map(to: A)

  A[0]=200; A[1]=200;

  #pragma omp target  //ref count->no copy
  {A[0]++; A[1]++;}


  #pragma omp target exit data map(from: A)

  cout << A[0]<< "  " << A[1] <<endl; //11,12
}
```

C++

```
program main


  integer :: A(2)=(/10,11/)

  !$omp target enter data map(to: A)

  A(1)=200; A(2)=200;

  !$omp target    !!ref count->no copy
    A(1)=A(1)+1; A(2)=A(2)+1
  !$omp end target


  !$omp target exit data map(from: A)

  print*, A(1), A(2)  !! 11, 12
end program
```

target update -- simple example

C++

```
#include <iostream>
using namespace std;

int main(){
 int A[2]={10,11};

 #pragma omp target data map(A)
 {
  {A[0]=200; A[1]=200;}
  #pragma omp target update to(A)

  #pragma omp target  //ref count->no copy
  {A[0]++; A[1]++;}


 }

 cout << A[0]<< "  " << A[1] <<endl; //11,12

}
```

FORTRAN

```
program main


 integer :: A(2)=(/10,11/)


 !$omp target data map(A)

 A(1)=200; A(2)=200;
 !$omp target update to(A)

 !$omp target    !!ref count->no copy
   A(1)=A(1)+1; A(2)=A(2)+1
 !$omp end target

 !$omp end target data


 print*, A(1), A(2)  !! 11, 12

end program
```

Directives for directional data movement: syntax

- map data motion restricted to one direction
- employs reference counter

(async with nowait clause)

target_enter_data map(type: vars) types: to, alloc

target_exit_data map(*type*: *vars*) types: from (release, delete)

- Similar to target_enter/exit_data, but
- updates without reference counter.

target update to(*vars*)

target update from(*vars*)

to and from
are the data motion clauses

Compiling code with OpenMP offloading statements

NVHPC <nvc, nvc++, nvfortran>

```
<compiler> -fopenmp -O3 -mp=gpu
```

LLVM <clang, clang++, flang>

```
<compiler> -fopenmp -O3 -fopenmp-targets=nvptx64
```

GNU <gcc, g++, gfortran>

```
<compiler> -fopenmp -O3 -foffload=nvptx-none
```

LAB I – basic offload

- Create interactive session on gpu compute node.
- Compile and run “hello from gpu” code.
- Compile and run target code block/function for single thread.
- Optimize code for efficient data transfer.
- Compare parallel axpy execution on host and device
- Instrument offloading code to run asynchronously.

```
ssh vista.tacc.utexas.edu
tar –xvf ~train00/openmp_offload_2025.tar
cd openmp_offload
```

- Login to Vista
- Untar files and cd to lab directory
- **Read top directory Instructions** file, cd into code directories and **follow instructions in README.md** file.

Agenda

- Map Review and Storage

common ways to change the device data (environment)

Evaluate on your own.

C/C++

```
int scal = 1;
 #pragma omp target data map(tofrom: scal)   //  scal is mapped for region
 {
     scal=10;
     #pragma omp target map(always,tofrom: scal) // make device/host consistent (10)
     { scal++;                          // mapped scal =  11
     }                               // cp back scal = 11

     scal++;
     #pragma omp target // default: scal is FIRSTPRIVATE, uses Host scal=12
     { scal++;        // 13
     }            // 12 scal Not Copied back

     scal=100;
     #pragma omp target update to(scal)   // consistent scal(100 on host & device)
     scal=500;
     #pragma omp target map(scal)        // map ignored
     {   scal++;                     // 101 Increased 100 to 101 on Device
     }                           // 500, Not Copied back
 }                               // 101 Copied back, same on Host and Device
```

common ways to change the device data (environment)

Evaluate on your own.

FORTRAN

```
integer :: scal = 1
 !$omp target data map(tofrom: scal)       !!  scal is mapped for region

    scal=10
    !$omp target map(always,tofrom: scal) !! make device/host consistent (10)
      scal=scal+1                          !! mapped scal =  11
    !$omp end target

    scal=scal+1
    !$omp target       !! default: scal is FIRSTPRIVATE, uses Host scal=12
      scal=scal+1     !! 13
    !$omp end target   !! 12 scal Not Copied back

    scal=100
    !$omp target update to(scal)   !! consistent scal(100 on host & device)

    scal=500
    !$omp target map(scal)        !! map ignored
      scal=scal+1               !! 101 Increased 100 to 101 on Device
    !$omp end target            !! 500, Not Copied back

 !$omp end target data           !! 101 Copied back, same on Host and Dev
```

CAUTION

device-only storage -- like cuda, uses device pointer

C/C++

```
int main(){
  double Val, *d_B, *d_C;    // (d_ = device pointers)

  int dev_id  = omp_get_default_device();         // device id number
  d_B=(double *)omp_target_alloc(N*N*sizeof(double),dev_id);
  d_C=(double *)omp_target_alloc(N*N*sizeof(double),dev_id);

  #pragma omp target map(from: Val) is_device_ptr(dev_B,dev_C) device(dev_id)
  {                              //specify dev pointers & their device

    for(int i=0; i<N; ++i)   // B,C storage used (no data transfer)
    for(int j=0; j<N; ++j)
      {d_B[i*N+j]=i*j; d_C[i*N+j]=i*j+1;}
    ...
    Val=3.14;
  }
  omp_target_free(dev_B, dev_id);
  omp_target_free(dev_C, dev_id);
}
```

alloc maptype for temporary space

C/C++

```
float x[N], y[N];
...
#pragma omp target ... map(alloc: x) map(tofrom: y)
{
  for(int i=0;i<n;i++) x[i]=sin(i%4)*cos(i%6);

  for(int i=0;i<n;i++) y[i]=a*x[i]+y[i];
}
```

Agenda

OpenMP Offloading

- Launch and direct device execution with target.
- Explicit/Implicit mapping ("transfer") to/from device.
- teams, distribute, and loop.
- Data Persistence: target data.
  Data Movers: target enter/exit data, target update.

■ declare target (ensure function/data execution/access on device).

Asynchronous Offloading with nowait and depend.

declare target – declare functions as device procedures

- Procedures in the same “compile” unit are automatically compiled for devices.
- “Externally” compiled procedures must be declared for target devices.

C/C++

```
void fun();

int main(){
  #pragma omp target
  fun();
}
```

C/C++

```
#include <omp.h>              // Difference Compile Unit
#include <stdio.h>

void fun(){
  printf("%d\n", omp_is_initial_device()); // 0 on GPU, 1 on HOST
}
#pragma omp declare target (fun)          // Compile for GPU, too.
```

declare target -- Multiple ways to do it (global vars, too)

C/C++

```
#pragma omp begin declare target

  static float a_global=2.0f;
  void   foo(){printf("external compiled unit\n");}

#pragma omp   end declare target
```

This is for declaring everything within a region

C/C++

```
static float a_global=2.0f;
void   foo(){printf("external compiled unit\n");}

#pragma omp declare target  (a_global, foo)
```

This is for declaring specific vars and funcs.

C/C++

```
static float a_global=2.0f;
void   foo(){printf("external compiled unit\n");}

#pragma omp declare target to(a_global, foo)
```

Same as above, using clause (***to*** is default).

declare target – declare functions as device procedures

- Procedures in the same “compile” unit are automatically compiled for devices.
- “Externally” compiled procedures must be declared for target devices.

F

```
program main
 use functions
 !$omp target
   call fun()
 !$omp end target
end program
```

F

```
module functions        !! Different ”Compile” unit.
 use omp_lib
 contains
 subroutine fun()
   !$omp declare target (fun)      !! Compile for GPU, too.
   print*, omp_is_initial_device() !! F on GPU, T on HOST
 end subroutine
end module
```

declare target -- Fortran

F90

```fortran
module functions
contains
 subroutine fun()
  !$omp declare target to(fun)
  print*, "externally compiled"
 end subroutine
end module
```

```fortran
program main
 !$omp target
  call fun()
 !end target
end program
```

F90

```fortran
module functions
 integer,parameter  :: N=8
 real               :: v(N)
 contains
  subroutine fun(one)
  integer, intent(in) :: one
   !$omp declare target enter(fun,v)
   print*, "externally compiled"
   v(1:N)=one
 end subroutine
end module
```

Include immediately after specifications.

List function and global variables in to/enter clause.

```fortran
program main
 use functions
 integer :: one=1
 !$omp target
  call fun(one)
 !$omp end target
 !$omp target update from(v)
 print*,v
end program
```

declare target syntax

#pragma omp begin declare target

//procedures & static variables here

#pragma omp end declare target

C/C++ only
6.? Fortran

#pragma omp declare target (*ext_list*)

Use !$omp for F90

ext_list: list of procedures(pcr) and/or static variables(var)

#pragma omp declare target *clauses*

Use !$omp for F90

clauses:

```
to/enter(ext_list)   avail. entire prog
link(    var_list)   avail. when mapped
device(      expr)   device specific (id)
indirect(prc_list)   for func. pointers
```

to(<v5.2)   enter(>=v5.2)

Asynchronous Offloading

```
ndev=omp_get_num_devices(); // Number of devices, ids=0,…,ndev-1                    C/C++
n = N/ndev;

 #pragma omp parallel num_threads(ndev)
 #pragma master
 {
   for(int id=0; id<ndev; id++)
   {                              //simple,unblanced partitioning
      ib = id*n; ie = ib+n;
      if(id==ndev-1){ ie=N; n=N-ib;}                 //final iter.
      #pragma omp target teams distribute parallel for \
            nowait device(id) map(y[ib:n],x[ib:n])   //ib=offset,n=sz
       for(int i=ib; i< ie; i++) y[i] = a * x[i] + y[i];
   }
 }                    // parallel barrier waits for tasks to complete
```

Could do this without *parallel master*, but use a *taskwait* at end to synchronize.

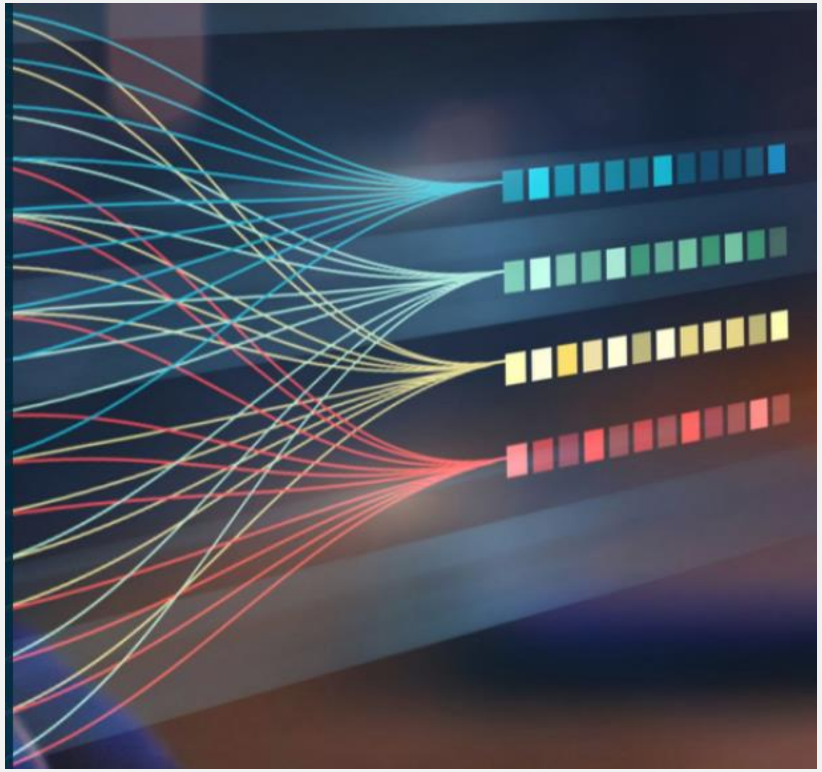
Asynchronous Offloading

```
ndev=omp_get_num_devices()       !! Number of devices, ids=0,...,ndev-1
n = NT/ndev                                                                          F

 !$omp parallel num_threads(ndev)
 !$omp master
   do id=0,ndev-1
                                  !!partitioning: simple,unblanced
     ib = 1 + id*n; ie = ib + n-1
     if(id == ndev-1) then; ie=NT;  endif    !!final iter.

     !$omp  target teams distribute parallel for &
     !$omp&       nowait device(id) map(y(ib:ie),x(ib:ie))
       do i=ib,ie;  y(i) = a * x(i) + y(i);  enddo
   enddo
 !$omp end master
 !$omp end parallel     !! parallel barrier waits for tasks to complete
```

begin
end
Fortran
Array
Syntax

Could do this without parallel master, but use a taskwait at end to synchronize.

Extras



Map: Reference count

- On entry to device environment:

Atomic Operation

  - If a corresponding list item is not present in the device data environment, then:
    - A new list item corresponding to original list item (on host) is created in the device data environment;
    - The corresponding list item has a reference count that is initialized to zero; and
    - The value of the corresponding list item is undefined;
  - If ref count is not incremented due to map clause, it is incremented by 1

- On exit from device environment:

Atomic Operation

  - if map-type is **delete** ref count is set to 0
  - if map-type is not **delete** the ref count is decremented by 1 (min 0)
  - If the reference count is zero then the corresponding list item is removed from the device data environment.

Thanks to Henry Jin (JPL) and Swaroop (Oak Ridge) and (CSC/EuroHPC JU) for slide and content collaboration.

Direct Device Allocation: omp_target_alloc() + is_device_ptr

C/C++

```
#include <omp.h>
int main(){
int a=2, N=1<<4; // 16
 int dev_no = omp_get_default_device();

 int *y  =(int*)         malloc(N*sizeof(N) );
 int *x_d=(int*)omp_target_alloc(N*sizeof(N), dev_no);

  for(int i=0;i<N;i++){ y[i]=1; }

  #pragma omp target is_device_ptr(x_d)
  for(int i=0;i<N;i++) x_d[i]=1;           //INIT

  #pragma omp target is_device_ptr(x_d) \
                   map(tofrom: y[0:N])
  for(int i=0;i<N;i++) y[i]=a*x_d[i]+y[i]; //AXPY

  omp_target_free(x_d, dev_no);
}
```

omp_target_alloc syntax

Syntax: void* omp_target_alloc(size_t size, int device_num);

returns the device address of
a storage location of size bytes

device_num : less than the result of
omp_get_num_devices() or the result of a call to
omp_get_initial_device().

```
  int init_dev =omp_get_initial_device();
  int    ndevs=omp_get_num_devices();
  cout<<"init_dev#= "<<init_dev <<" # non-host_devs"<< ndevs<<endl;

OUTPUT: init_dev#= 3 # non-host_devs3
```

Direct memory copy with omp_target_memcpy

C/C++

```
void get_dev_cos(double *mem, int s){
   int h, t, i;
   double * mem_dev_cpy;
   h = omp_get_initial_device();
   t = omp_get_default_device();

   mem_dev_cpy = (double *)omp_target_alloc( sizeof(double) * s, t);

                        /* dst  src */
   omp_target_memcpy(mem_dev_cpy, mem, sizeof(double)*s,
                            0,    0,
                            t,   h);

   #pragma omp target is_device_ptr(mem_dev_cpy) device(t)
   #pragma omp teams distribute parallel for
     for(i=0;i<s;i++){ mem_dev_cpy[i] = cos((double)i); } /* init data */

                 /* dst  src */
    omp_target_memcpy(mem, mem_dev_cpy, sizeof(double)*s,
                    0,          0,
                    h,          t);
    omp_target_free(mem_dev_cpy, t);
}
```

Verifying OpenMP Offloading

- When the (generic) OpenMP flag is on (e.g. usually –fopenmp) and a compiler doesn’t understand the target compiler flag it **defaults to fallback execution of the region on the host**.

How to check if target is offloading to GPU.

C/C++

```
#include <stdio.h>
#include   <omp.h>
int main(){
 #pragma omp target
 {printf("%d\n", omp_is_initial_device());} // 0 run on GPU, 1 run on HOST
}
```

Fortran

```
program main
 use omp_lib
 !$omp target
   print*, omp_is_initial_device()          !! F run on GPU, T run on HOST
 !$omp end target
end program
```

Verifying OpenMP Offloading

Useful offloading test.

C/C++

```
#include <stdio.h>
#include   <omp.h>
int main(){
  #pragma omp target
  {printf(”ranOnGPU=%c T=true\n", (omp_is_initial_device()==0) ? 'T':'F');}
}
```

Fortran

```
program main
  use omp_lib
  !$omp target
     print*, ”ranOnGPU= ", .not. omp_is_initial_device(), " T=TRUE, F=FALSE"
  !$omp end target
end program
```