# Fundamentals of Accelerated Programming with OpenMP

GPU Programming Workshop | LRZ | 8 – 12 June 2026

# Agenda

- Introduction
- OpenMP Basics
  - Lab 1: Profiling with Nsight Systems  (similar to Day 1)
  - Lab 2: OpenMP Directives
- OpenMP Offloading: Basics
  - Lab 3: GPU Programming using OpenMP
- OpenMP Offloading: Optimising Data Transfers
  - Lab 4: Data Management

# Two Paradigms for Parallel Programming

- **OpenMP Standard**
  - OpenMP 1.0 in 1997 (Fortran) / 1998 (C, C++)
  - OpenMP 3.0 (May 2008)
    - tasking etc.
  - OpenMP 4.0 (July 2013)
    - SIMD, affinity policies, accelerator support
  - OpenMP 5.0 (Nov 2018)
    - two new tool interfaces, multilevel memory systems
  - OpenMP 6.0 (Nov 2024)
    - improvements in usability and fine grain control
- **Base Languages**
  - Fortran
  - C, C++
- **Resources**
  - http://www.openmp.org

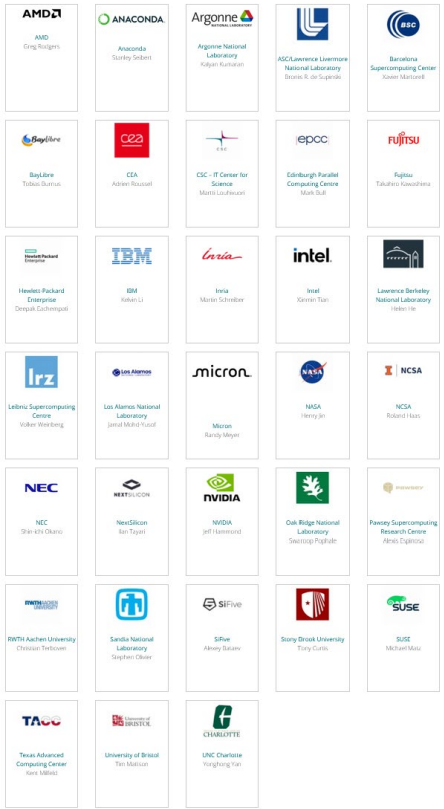
# OpenMP Standard

## OpenMP Standard



https://www.openmp.org/specifications/

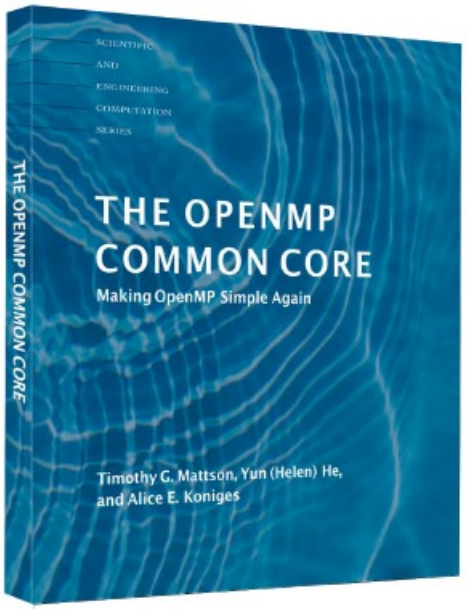
# OpenMP Architecture Review Board (ARB)




The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

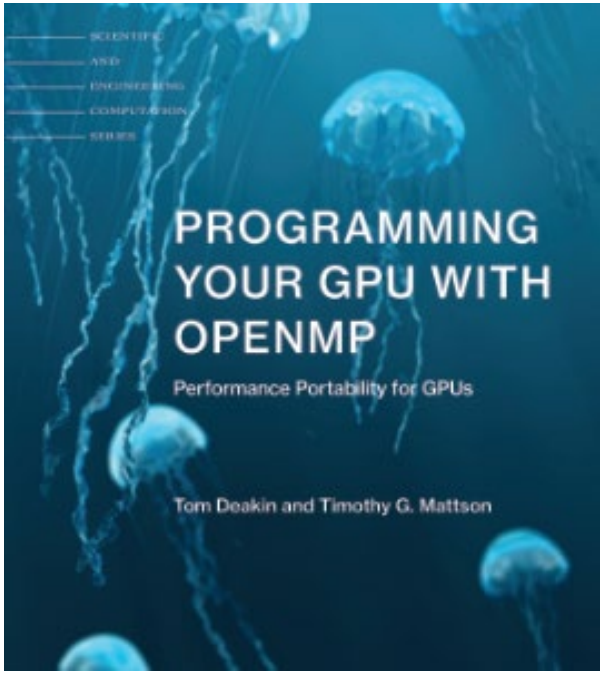
# Recent Books about OpenMP


Covers all of the
OpenMP 4.5 features, 2017




Introduces the
OpenMP Common Core, 2019

# Recent Books about OpenMP

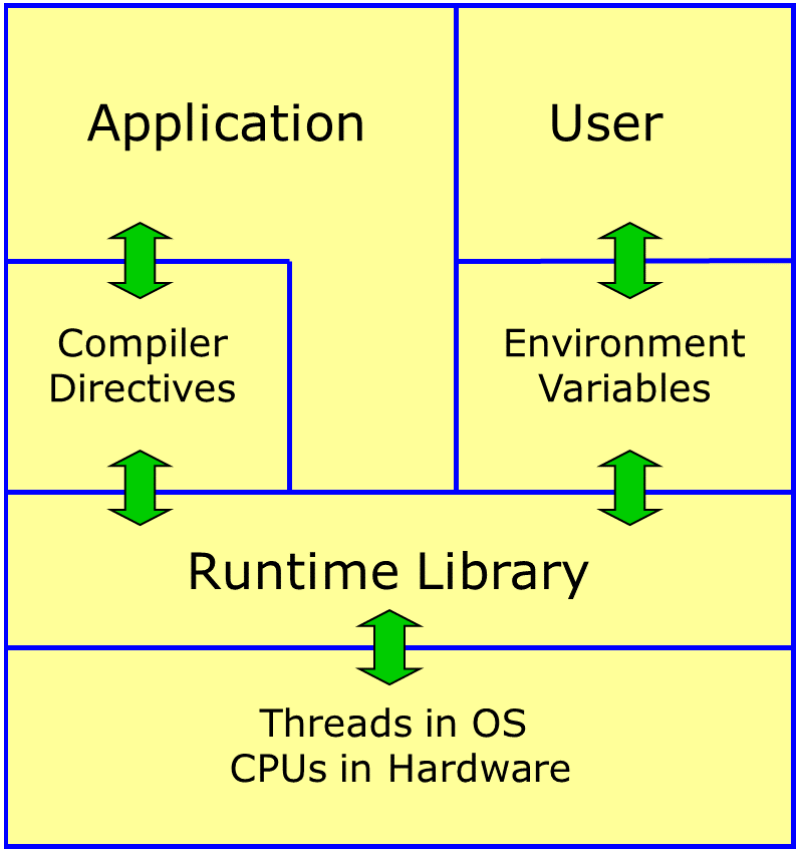



Covers all about Accelerator Programming, 2023

# OpenMP Basics

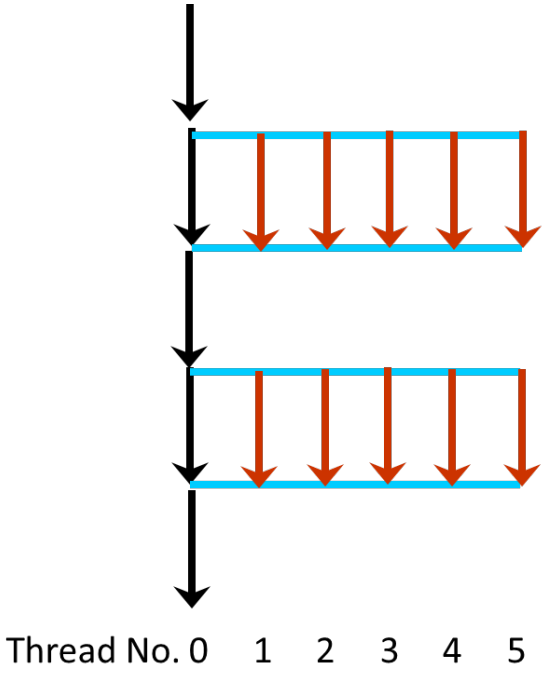
Reinhold Bader (LRZ)
Georg Hager (NHR@FAU)
Volker Weinberg (LRZ)

# OpenMP Programming Model




- **Operating system view:**
  - parallel work done by **threads**
- **Programmer's view:**
  - **directives**: comment lines in code, e.g.
    - `!$omp parallel`
    - `#pragma omp parallel`
  - library routines, e.g.
    - `omp_get_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
- **User's view:**
  - **environment variables** determine: resource allocation, scheduling strategies and other (implementation-dependent) behaviour, e.g.
    - `OMP_NUM_THREADS`
    - `OMP_SCHEDULE`
    - `OMP_NESTED`

# OpenMP Programming Model




- Program start: only initial thread (formerly known as master thread) runs
- Parallel region: team of worker threads is generated (“fork”)
- Threads synchronize when leaving parallel region (“join”)
- Only initial thread executes sequential part (worker threads persist, but are inactive)
- Task and data distribution possible via directives
- Nesting of parallel regions:
  - allowed, but level of support implementation dependent
- Usually optimal:
  - one thread per processor core
  - other resource mappings are allowed/possible

# Parallel region: Simplest Program Example: C/C++

```
#include <stdio.h>
#include <omp.h>

int nthr, myth;

int main(int arc, char *argv[])

{

#pragma omp parallel private(myth)

  {

    #pragma omp single

    nthr = omp_get_num_threads();


    myth = omp_get_thread_num();

    printf("Hello from %i of %i\n", myth, nthr);

  }

}
```

- **Parallel region directive:**
  - enclosed code executed by **all** threads
  - may include subprogram calls („dynamic region“)
- **Special function calls:**
  - Include file <omp.h>
  - here: get number of threads and index of executing thread
- **Data scoping:**
  - uses a **clause** on the directive
  - `myth` thread-local: **private**
  - `nthr` process-global: **shared**

(will be discussed in more detail later)

# Parallel region: Simplest Program Example: Fortran

```
program hello
  use omp_lib
  implicit none
  integer :: nthr, myth

!$omp parallel private(myth)

!$omp single
  nthr = omp_get_num_threads()
!$omp end single

  myth = omp_get_thread_num()

  write(*,*) "Hello from ", myth, "of ", nthr

!$omp end parallel

end program hello
```

- **Parallel region directive:**
  - enclosed code executed by **all** threads
  - may include subprogram calls („dynamic region“)
- **Special function calls:**
  - module **`omp_lib`** provides interface
  - here: get number of threads and index of executing thread
- **Data scoping:**
  - uses a **clause** on the directive
  - **`myth`** thread-local: **private**
  - **`nthr`** process-global: **shared**

(will be discussed in more detail later)

# Compiling and Running an OpenMP Program

**Compile Fortran (e.g. with NVIDIA compiler):**

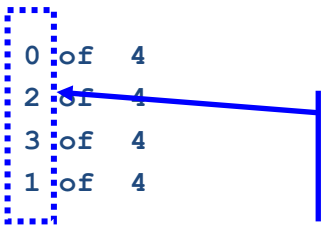
```
nvfortran -mp=multicore -Minfo=mp,opt
   -o hello.exe hello.f90
```

**Compile C (e.g. with NVIDIA compiler):**

```
nvc -mp=multicore -Minfo=mp,opt
   -o hello.exe hello.f90
```

**Run:**

```
export OMP_NUM_THREADS=4
./hello.exe
 Hello from  0 of  4
 Hello from  2 of  4
 Hello from  3 of  4
 Hello from  1 of  4
```



ordering not reproducible

- **OpenMP environment**
  - defines runtime behaviour
  - here: number of threads used

# OpenMP C/C++ Syntax

- Include file:

```
#include <omp.h>
```

- Preprocessor directive: uses pragma feature

```
#pragma omp <directive> [clause …]
```

- Conditional compilation: OpenMP switch sets preprocessor macro

```
#ifdef _OPENMP
  … /* do something */
#endif
```

- Continuation line:

```
#pragma omp directive \
   clause
```

# OpenMP Fortran Syntax

- Specifications:
  - Fortran 77 style

```
include "omp_lib.h"
```

  - Fortran 90 module (**preferred**)

```
use omp_lib
```

- Directives:
  - fixed form source:

```
C$OMP <directive> [<clause [(<args>)]>, …]
```

  - free form source (preferred):

```
!$OMP <directive> [<clause [(<args>)]>, …]
```

sentinel starting in column 1,
also : ***$OMP, !$OMP**

- Conditional compilation:

```
 myid = 0
!$ myid = omp_get_thread_num()
```

  - In fixed form also sentinels *$, c$
- Continuation line:

```
!$OMP <directive> &
!$OMP <clause>
```

# OpenMP Syntax: Remarks on Clauses

- Many (but not all) OpenMP directives support clauses
  - more than one may appear on a given directive
- Clauses specify additional information associated with the directive
  - modification of directive's semantics
- "Simplest example" from above:
  - `private(…)` appears as clause to the `parallel` directive
- The specific clause(s) that can be used depend on the directive

# OpenMP Syntax: Structured Block

- **Defined by braces in C/C++**
- **In Fortran:**
  - code between begin/end of an OpenMP construct must be a complete, valid Fortran block
- **Single point of entry:**
  - no `GOTO` into block (Fortran), no `setjmp()` to entry point (C)
- **Single point of exit:**
  - `RETURN, GOTO, EXIT` outside block are prohibited (Fortran)
  - `longjmp()` and `throw()` must not violate entry/exit rules (C, C++)
  - **exception:** termination via `STOP` or `exit()`

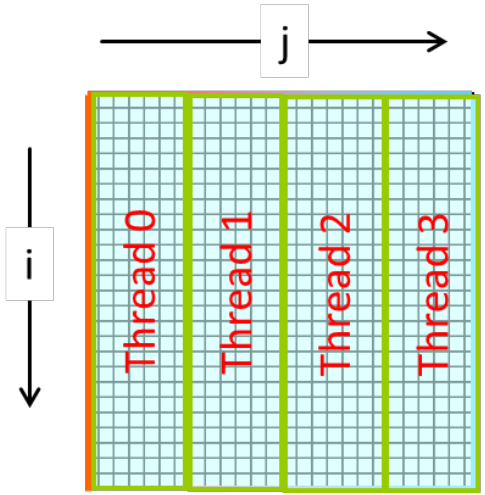
- **Block structure example:**
  - C version of simplest program

```
#include <omp.h>

int main() {
  int numth = 1;
#pragma omp parallel
  {
     int myth = 0; /* private */
#ifdef _OPENMP
#pragma omp single
      numth = omp_get_num_threads();
      /* block above: one statement */
      myth = omp_get_thread_num();
#endif
      printf("Hello from %i of %i\n",\
             myth,numth);
  }  /* end parallel */
}
```

# Work Sharing in OpenMP (1): C/C++

- **Making parallel regions useful …**
  - divide up work between threads
- **Example:**
  - working on an array processed by a nested loop structure




- iteration space of **directly nested loop** is sliced

```
float a[ndim][ndim];

int main(int arc, char *argv[])
{
#pragma omp parallel
  {
#pragma omp for
    for(int j=0;j<ndim;j++) {
      for(int i=0;i<ndim;i++) {
        a[i][j]= …;
      }
    }
  …
  }
}
```

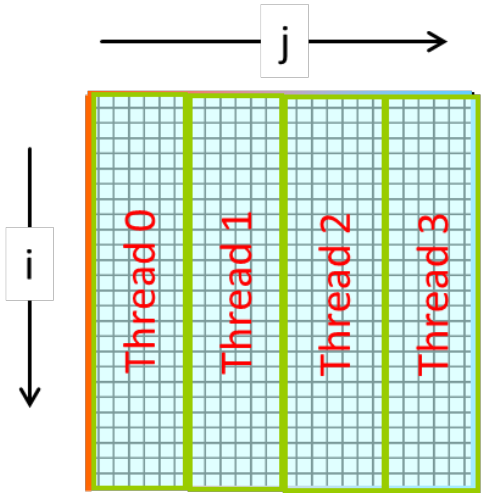
j-loop is sliced

synchronization between threads

further parallel execution

# Work Sharing in OpenMP (1): Fortran

- **Making parallel regions useful …**
  - divide up work between threads
- **Example:**
  - working on an array processed by a nested loop structure




```
real :: a(ndim, ndim)
…
!$omp parallel
!$omp do
do j=1, ndim
  do i=1, ndim
    …
    a(i, j) = …
  end do
end do
!$omp end do
  …
!$omp end parallel
```

j-loop is sliced

synchronization between threads

further parallel execution

- iteration space of **directly nested loop** is sliced

# Work Sharing in OpenMP (2)

- **Synchronization behaviour:**
  - all threads (by default) **wait for completion** at the end of the work sharing region („barrier“)
  - following references using an array element by **other** threads are therefore OK.
- **Slicing of iteration space:**
  - „loop scheduling“
  - default behaviour is implementation dependent
  - usually as equal as possible chunks of largest possible size
- **Additional clauses on `!$OMP DO` / `#pragma omp for`** possible (discussed later)

- **Fortran syntax:**

```
!$omp do [clause]
do ...
   ...   // loop body
end do
```

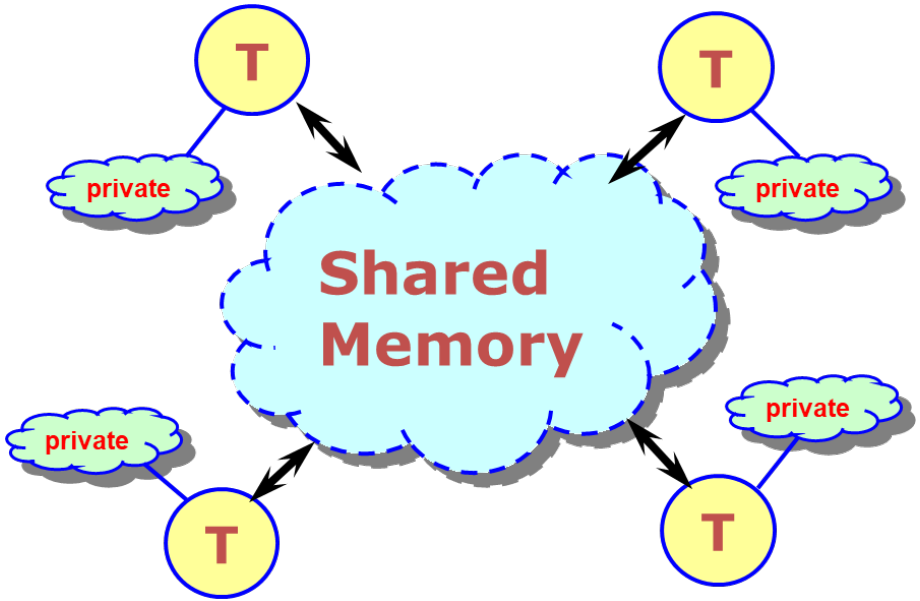
- **C/C++ syntax:**

```
#pragma omp for [clause]
for ( ... )  {
      ...   // loop body
}
```

- **Restrictions on loop structure:**
  - trip count must be **computable** at entry to loop
  - loop body with single entry and single exit point

# Memory Model

- **Two kinds of memory exist in OpenMP**




- **Threads access globally shared memory**
- **Data can be shared or private**
  - shared data – one instance of an entity available to all threads
  - private data – each per-thread copy only available to thread that owns it
- **Data transfer** transparent to programmer
- **Synchronization** takes place (is mostly implicit)
- **threadprivate variables**

# Data-Sharing Attributes

- By default most variables are shared
  - local variables outside the scope of construct
  - static/global (C/C++) or save/common (Fortran) variables

```
int s = 1;

#pragma omp parallel
{
  int p = omp_get_thread_num();
  printf("s=%d p=%d\n", s, p);
}
```

- Except
  - variables* defined inside the construct are private
    - i.e. declared inside `{}`-block or `BLOCK/END BLOCK`
  - variables* local to functions/routines called from within the region are private
  - loop iteration variables of worksharing loops are private

* non-static (C/C++) or without save attribute (Fortran)

# Data-Sharing Attribute Clauses

- Clauses for explicitly specifying how a variable should be treated
  - supported by several directives, e.g., `parallel`, `do`/`for`, `single`, `sections`, `task`, ...
- Clauses:
  - `shared(var1, var2, …)`
  - `private(var1, var2, …)`
  - private + special operation
    - `firstprivate(var1, var2, …)`
    - `lastprivate`, for `do`/`for` construct
- Change default:
  - Fortran `default(shared|private|firstprivate|none)`
  - C/C++: `default(shared|none)`
  - best practice: `default(none)`
    - every variable referenced must appear in a `shared`/`private`/… clause
    - avoids incorrect assumptions about shared/private

# Scoping: Second-Simplest Example: C/C++

- Summation inside a loop

```
 float s, stot;
 stot = 0.;
#pragma omp parallel private(s)
  {
    s = 0.;
#pragma omp for
    for(int i=0;i<ndim;i++) {
      … // workload
      s = s + … ;
    }
#pragma omp critical
    {
      stot = stot + s;
    }
  }
```

- **Note:** large workload inside loop improves threaded performance

- require thread-individual variable for partial sum calculated on each thread
- **but:** private copies of variables are **undefined** at entry to, and become **undefined** at exit of the parallel region
- **therefore:** collect partial sums to a **shared** variable defined after the worksharing region
- **updates** to shared variable must be specially protected:

→ use a **critical region**

→ only one thread at a time may execute (mutual exclusion)

(performance impact due to explicit synchronization)

# Scoping: Second-Simplest Example: Fortran

- Summation inside a loop

```
real :: s, stot
stot = 0.0
!$omp parallel private(s)
s = 0.0
!$omp do
do i=1, ndim
    … ! workload
    s = s + …
end do
!$omp end do

!$omp critical
  stot = stot + s
!$omp end critical

!$omp end parallel
```

- **Note:** large workload inside loop improves threaded performance

- require thread-individual variable for partial sum calculated on each thread
- **but:** private copies of variables are **undefined** at entry to, and become **undefined** at exit of the parallel region
- **therefore:** collect partial sums to a **shared** variable defined after the worksharing region
- **updates** to shared variable must be specially protected:

→ use a **critical region**

→ only one thread at a time may execute (mutual exclusion)

(performance impact due to explicit synchronization)

# `collapse` Clause

- Combines multiple loops into a single loop.
- The length of the resulting parallel loop is the combined length of the loops affected.
- The parameter `n` is used to specify how many loops should be collapsed.

- Syntax (C/C++)

```
#pragma omp for collapse (n) [clause[[,] clause],…]
for loops
```

- Syntax (Fortran)

```
!$omp do collapse (n) [clause[[,] clause],…]
do loops
!$omp end do
```

# Private Variables – Masking: C/C++

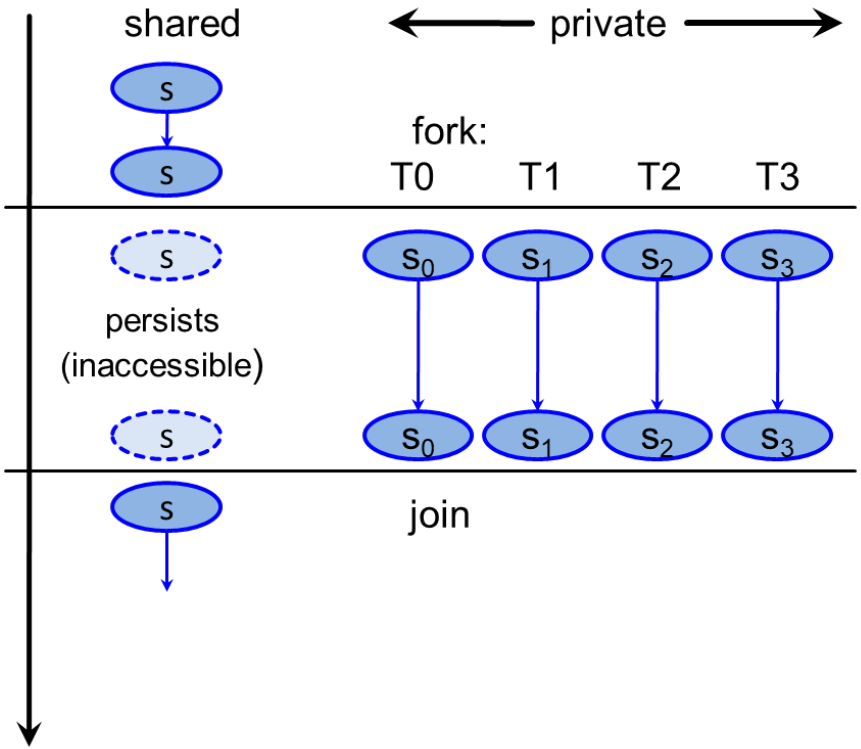
```
float s;

  s = … ;

#pragma omp parallel private(s)
{
  s = … ;
  … = … + s;
}

  … = … + s;
```

- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region



# Private Variables – Masking: Fortran

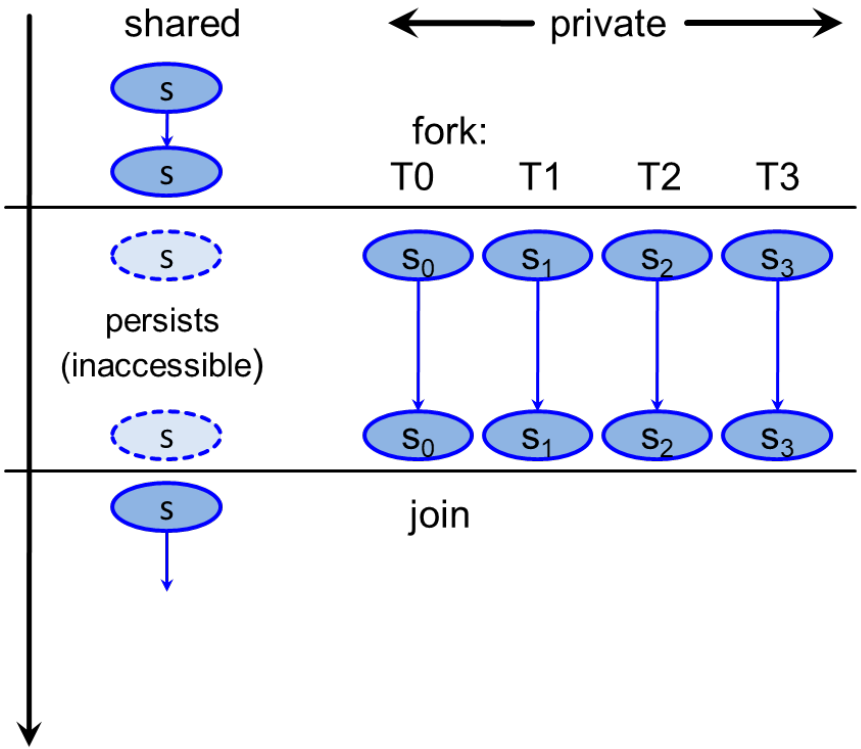
```
real :: s

  s = …

!$omp parallel private(s)
  s = …
  … = … + s
!$omp end parallel

  … = … + s
```

- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region



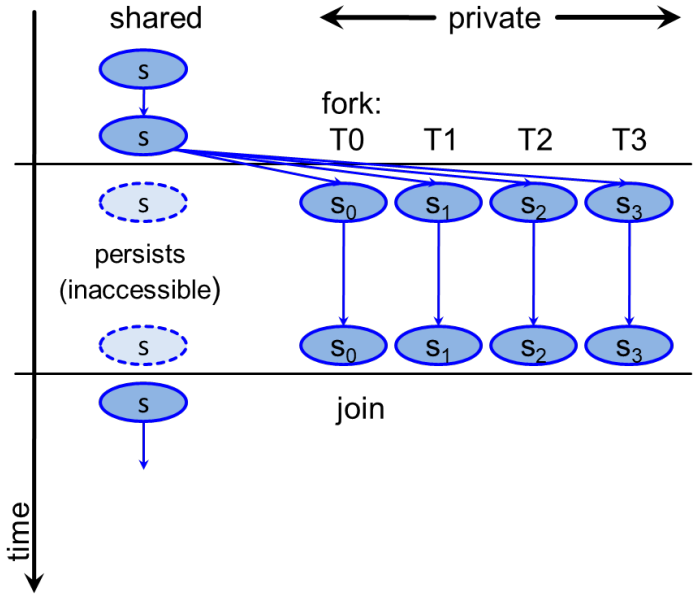
# The firstprivate Clause: C/C++

```
float s;

  s = … ;

#pragma omp parallel firstprivate(s)
{
  … = … + s;
}

  … = … + s;
```




- **Extension of private:**
  - value of master copy is transferred to private variables

# The firstprivate Clause: Fortran

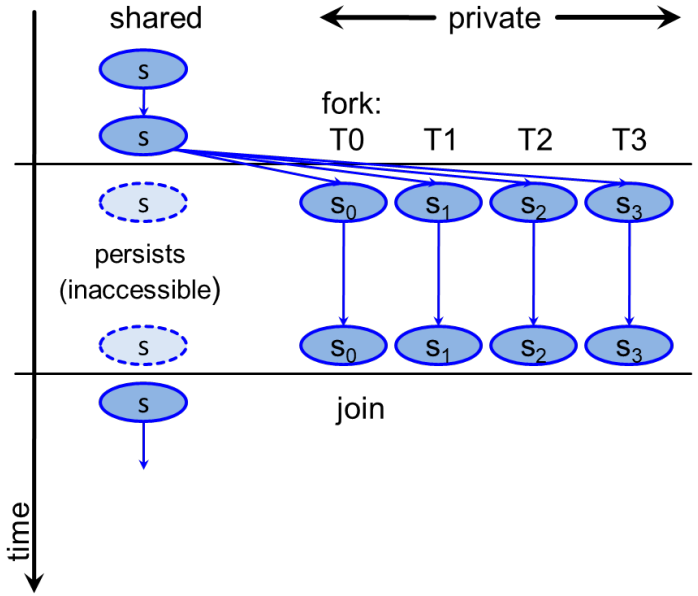
```
real :: s

  s = …

!$omp parallel firstprivate(s)

  … = … + s
!$omp end parallel

  … = … + s
```




- **Extension of private:**
  - value of master copy is transferred to private variables

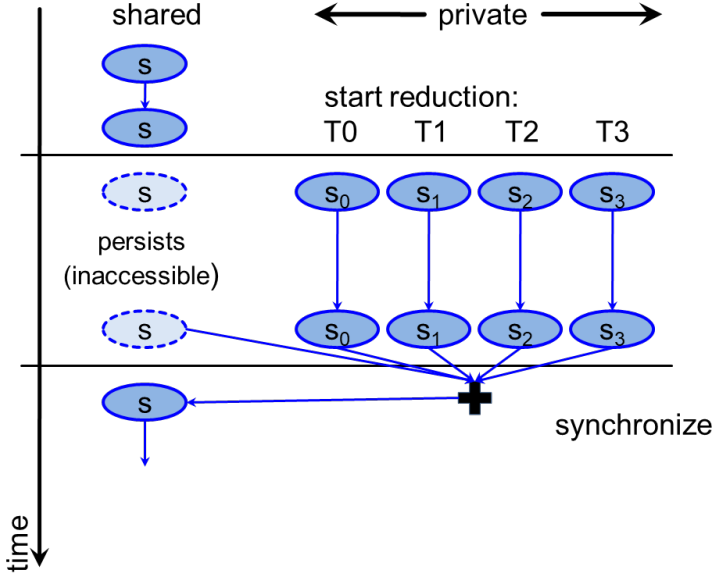
# Reduction Operations (1): C/C++

```
 float s;

#pragma omp parallel
  {
#pragma omp for reduction(+:s)
    for(int i=0;i<ndim;i++) {
      ...;
      s = s + …;
    }
     … = … * s;
  }
```

s is still shared here

consistency required




- **At synchronization point:**
  - reduction operation is performed
  - result is transferred to master copy

**Note:** this improves on the summation example (no explicit critical region needed)

# Reduction Operations (2): C/C++

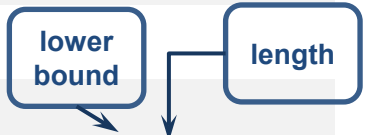
- **Initial value of reduction variable**
  - depends on operation

| Operation | Initial Value |
|---|---|
| + | 0 |
| - | 0 |
| * | 1 |
| & | ~ 0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |
| max | min(type) |
| min | max(type) |

- **Consistency required**
  - operation specified in clause vs. update statement

- **Multiple reductions:**
  - multiple scalars, or an array:

```
float x, y, z;
#pragma omp for reduction(+:x, y, z)
```




```
float a[n];
#pragma omp for reduction(*:a[0:n])
```

```
#pragma omp for reduction (+:a[0:n]) \
    reduction (*:b[0:n],c[0:n])
```

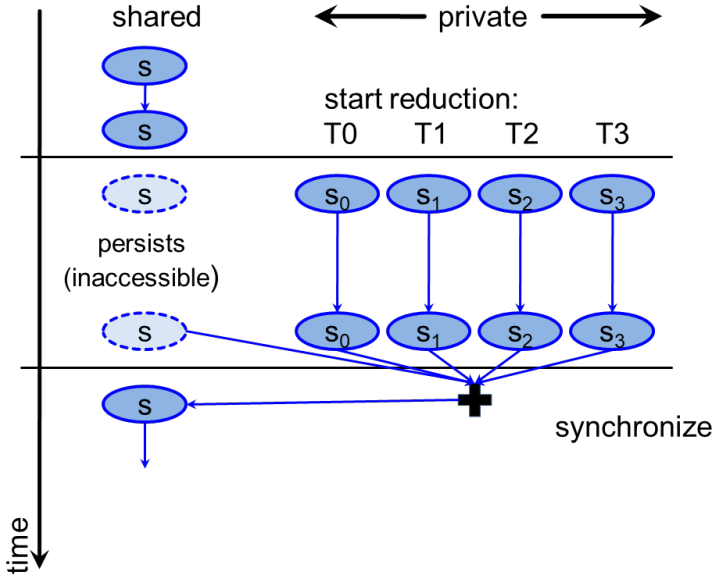
# Reduction Operations (1): Fortran

```
real :: s

!$omp parallel
!$omp do reduction(+:s)
  do i = …
    …
      s = s + …
  end do
!$omp end do
  … = … * s
!$omp end parallel
```

**s is still shared here**

**consistency required**




**Note:** this improves on the summation example (no explicit critical region needed)

- **At synchronization point:**
  - reduction operation is performed
  - result is transferred to master copy

# Reduction Operations (2): Fortran

- **Initial value of reduction variable**
  - depends on operation

| Operation | Initial Value |
|---|---|
| + | 0 |
| - | 0 |
| * | 1 |
| .and. | .true. |
| .or. | .false. |
| .eqv. | .true. |
| .neqv. | .false. |
| MAX | min(type) |
| MIN | max(type) |
| IAND | all bits set |
| IEOR | 0 |
| IOR | 0 |

- **Consistency required**
  - operation specified in clause vs. update statement

- **Multiple reductions:**
  - multiple scalars, or an array:

```
real :: x, y, z
!$OMP do reduction(+:x, y, z)
```

```
real :: a(n)
!$OMP do reduction(*:a)
```

```
!$OMP do reduction(+:x, y) &
!$OMP    reduction(*:z)
```

# Hands-On Session

## Lab 1: Profiling with Nsight Systems (similar to Day 1)

## Lab 2: OpenMP Directives

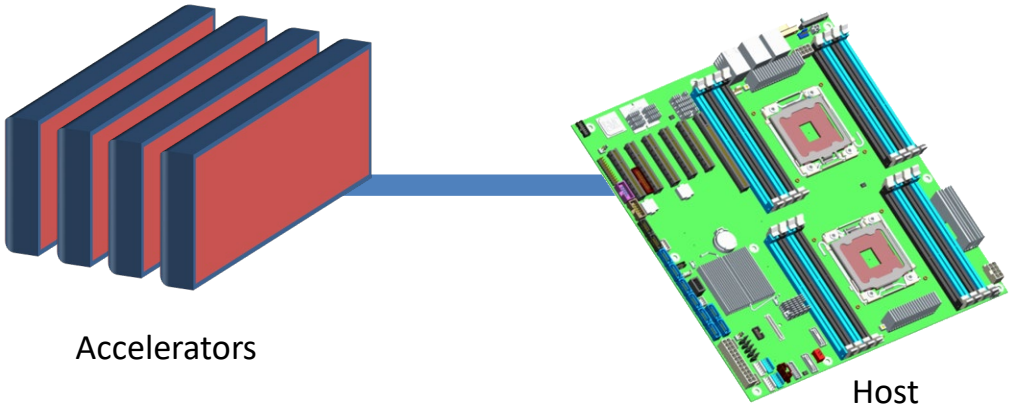
| C | Fortran |
| --- | --- |
| Compile with: | Compile with: |
| `nvc -fast -mp=multicore -Minfo=mp,opt` | `nvfortran -fast -mp=multicore -Minfo=mp,opt` |

# OpenMP Offloading

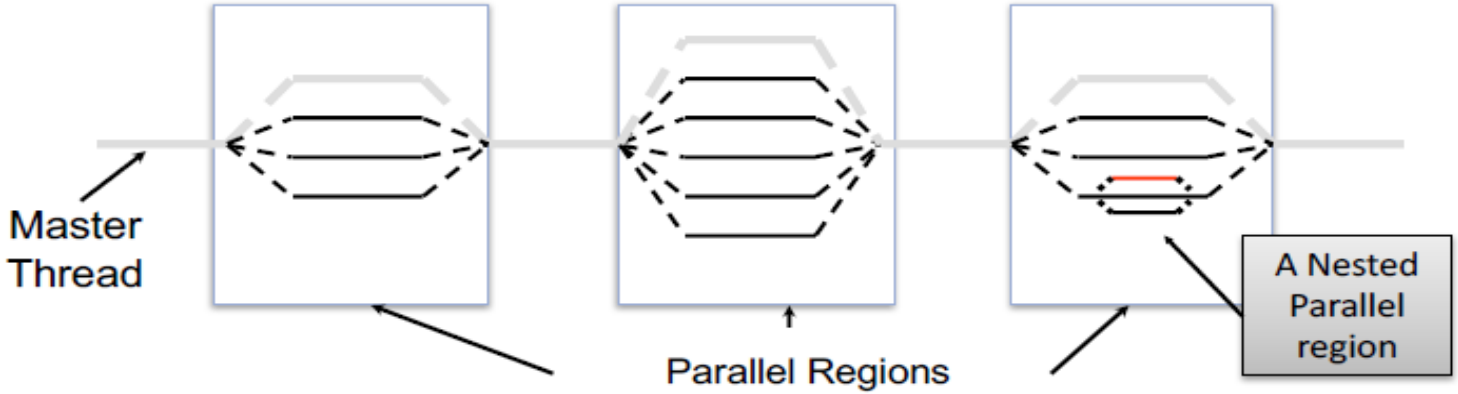
Based on material by Dr.-Ing. Michael Klemm

# Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading



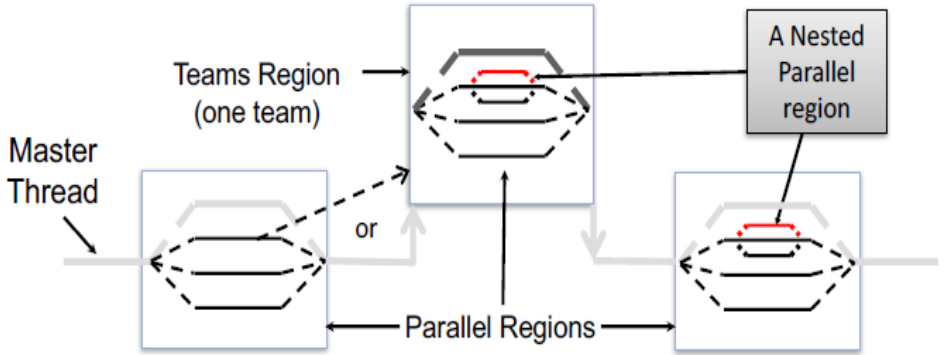
# Execution Model before OpenMP 4.x



# OpenMP 4.x Execution Model

- Create and destroy threads,
- create and destroy  leagues of thread teams,
- assign / distribute work (tasks) to threads and devices,
- specify which data is shared and which is private,
- specify which data must be available to the device,
- coordinate thread access to shared data.

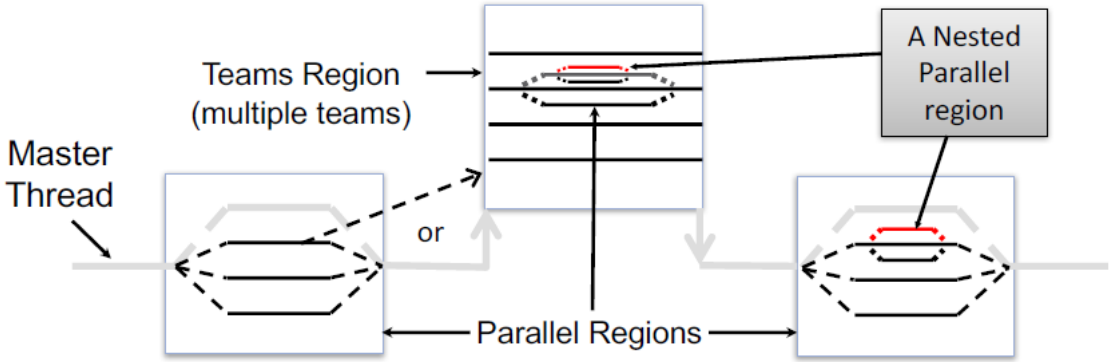
Execution model after OpenMP 4.x executing one team:



# OpenMP 4.x Execution Model

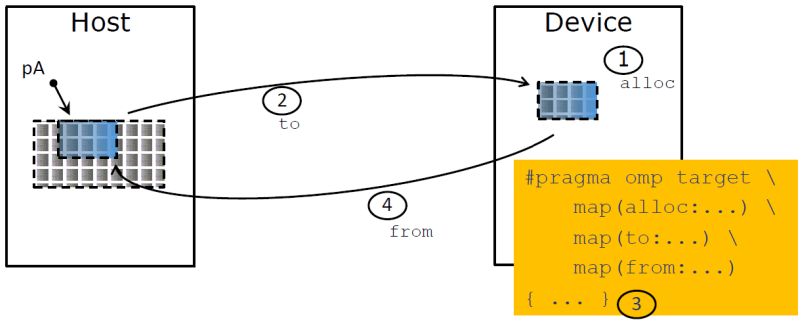
- Create and destroy threads,
- create and destroy  leagues of thread teams,
- assign / distribute work (tasks) to threads and devices,
- specify which data is shared and which is private,
- specify which data must be available to the device,
- coordinate thread access to shared data.

Execution model after OpenMP 4.x executing one team:



# OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload data from the host to the target device at the opening curly brace.
    - Download data from the target device at the closing curly brace.



# OpenMP for Devices - Constructs

- Transfer control and data from the host to the device
- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

# Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    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);
}
```

Timing code

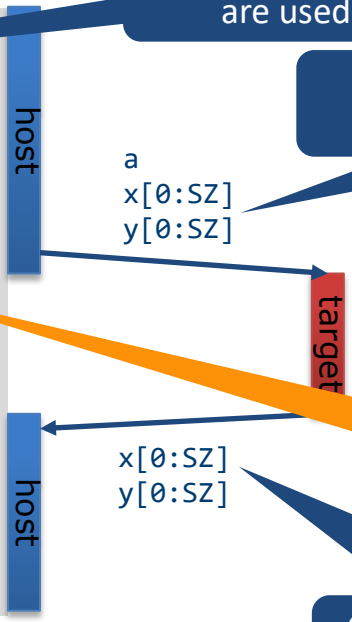
This is the code we want to execute on a target device (i.e., GPU)

Timing code

Don’t do this at home!
Use a BLAS library for this!

# Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target "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);
}
```




The compiler identifies variables that are used in the `target` region.

All accessed arrays are copied from host to device and back

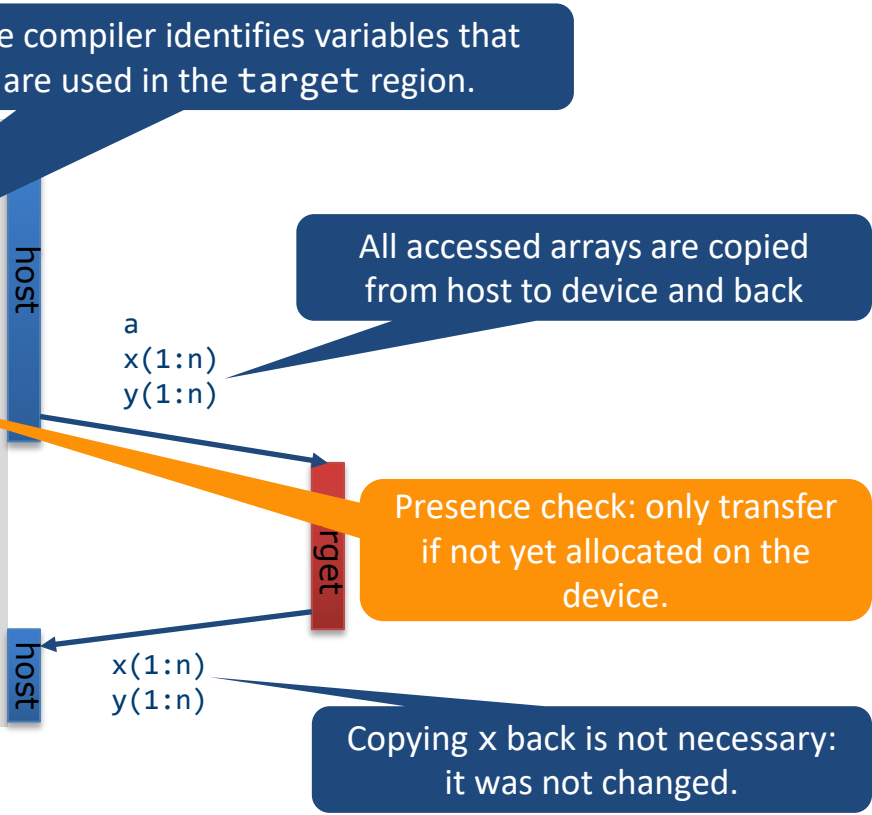
Presence check: only transfer if not yet allocated on the device.

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

# Example: saxpy

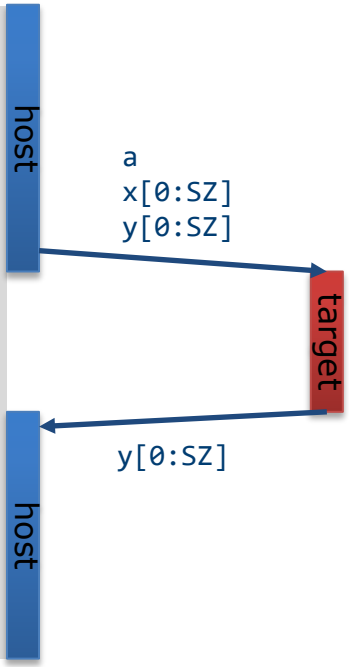
```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

!$omp target “map(tofrom:y(1:n))”
    do i=1,n
       y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```



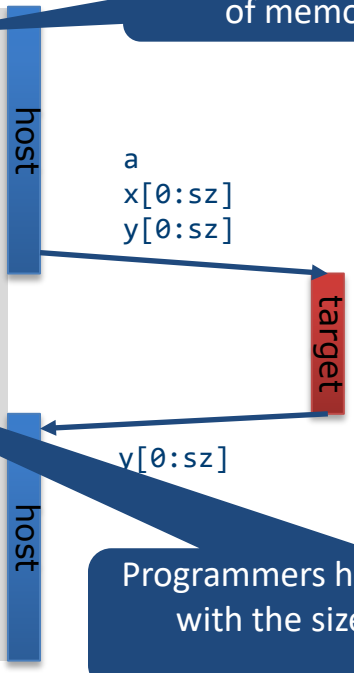
# Example: saxpy

```
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);
}
```



# Example: saxpy

```
void saxpy(float a, float* x, float* y,
           int 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);
}
```




The compiler cannot determine the size of memory behind the pointer.

Programmers have to help the compiler with the size of the data transfer needed.

# Mapping Clause

C

```
map(to: array[start_index:length])

map(from:array[start_index:length])

map(tofrom: array[start_index:length])

map(alloc: array[start_index:length])

map(release: array[start_index:length])

map(delete: array[start_index:length])
```

Fortran

```
map(to: array(start_index:end_index)

map(from: array(start_index:end_index))

map(tofrom: array(start_index:end_index))

map(alloc: array(start_index:end_index))

map(release: array(start_index:end_index))

map(delete: array(start_index:end_index))
```

For 2D arrays in Fortran, array sections must be specified as `array(start_index1:end_index1, start_index2:end_index2)`. If the whole array is used in OpenMP data mapping, it can be written simply as e.g. `map(to: array)` without specifying bounds in Fortran.

# Exploiting Multilevel Parallelism on the Target Device

- The `target` construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - This is intentional!

- OpenMP separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device
  - In theory, this can be combined with any OpenMP construct
  - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

# Example: saxpy

```
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.

# `teams` Construct

- Support multi-level parallel devices
- Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran):

```
!$omp teams [clause[[,] clause],…]
structured-block
```

- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

# `teams` and `distribute` Construct

- The teams construct creates a *league* of thread teams
  - Spawns one or more thread teams with the same number of threads
  - To better use the GPU resources, use many thread teams
  - The master thread of each team executes the **teams** region (redundantly)
  - The (max.) number of teams is specified by the **num_teams** clause
  - Each team executes with (max.) **thread_limit** threads
  - Threads in different teams cannot synchronize with each other

- Use the `distribute` directive to
  - Distribute the iterations of the next loop to the master threads of the teams
  - Does not generate parallelism/worksharing within the thread teams

# Multi-level Parallel saxpy

- Manual code transformation
  - Tile the loop into an outer loop and an inner loop.
  - Assign the outer loop to “teams”.
  - Assign the inner loop to the “threads”.
  - (Assign the inner loop to SIMD units.)

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

# Composite Constructs and Shortcuts

- omp distribute — Iterations distributed across the master threads of all teams in a teams region
  - **omp distribute** simd — dito + executed concurrently using SIMD instructions
  - **omp distribute** parallel for — executed in parallel by multiple threads that are members of multiple teams
  - **omp distribute** parallel for simd — dito + executed concurrently using SIMD instructions

- omp teams — creates a league of thread teams and the master thread of each team executes the region
  - **omp teams** distribute
  - **omp teams** distribute simd
  - **omp teams** distribute parallel for
  - **omp teams** distribute parallel for simd

- omp target — map variables to a device data environment and execute the construct on that device
  - **omp target** simd
  - **omp target** parallel
  - **omp target** parallel for
  - **omp target** parallel for simd

- omp target teams
  - omp target teams distribute
  - omp target teams distribute simd
  - omp target teams distribute parallel for
  - omp target teams distribute parallel for simd

# Multi-level Parallel saxpy

- For convenience, OpenMP defines composite constructs to implement the required code transformations

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

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted
!$omp omp target teams distribute parallel do simd map(to:x) map(tofrom:y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

# Runtime Support Routines

- **`omp_set_default_device(int dev_num)`**
  Sets the default device (e.g., GPU) used for OpenMP target regions when no device is explicitly specified.
- **`omp_get_default_device(void)`**
  Returns the ID of the currently selected default OpenMP device (e.g., GPU).
- **`omp_get_num_devices(void)`**
  Returns the total number of available OpenMP devices (e.g., GPUs) in the system.
- **`omp_get_num_teams(void)`**
  Returns the number of teams (GPU blocks) created in the current target teams region.
- **`omp_get_team_num(void)`**
  Returns the ID of the current team within a target teams region.
- **`omp_is_initial_device(void)`**
  Returns whether the code is currently executing on the host (initial device).
- **`omp_get_initial_device(void)`**
  Returns the device ID of the host (initial device)

# Runtime Support Routines

```c
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Number of devices: %d\n", omp_get_num_devices());
    printf("Initial device (host ID): %d\n", omp_get_initial_device());
    printf("Default device: %d\n", omp_get_default_device());

    // Set default device if at least one GPU exists
    if (omp_get_num_devices() > 0) {
        omp_set_default_device(0);
        printf("Default device set to: %d\n", omp_get_default_device());
    }

    int nteams = 4;
    printf("--- Start: Offloading ---\n");


#pragma omp target teams num_teams(nteams)
    {
        if (omp_get_team_num()==0) {
            printf("Running on initial device? %d\n", omp_is_initial_device());
            printf("Number of teams: %d\n", omp_get_num_teams());
        }
#pragma omp distribute parallel for
        for(int i=0;i<8;i++) {
            printf("Team %d Thread: %d\n", omp_get_team_num(), omp_get_thread_num());
        }
    }

    printf("--- Stop: Offloading ---\n");

    printf("Back on host: %s\n",
           omp_is_initial_device() ? "YES (HOST)" : "NO (GPU)");

    return 0;
}
```

```
b24n0003@a1721:~/test$ ./a.out
Number of devices: 1
Initial device (host ID): -1
Default device: 0
Default device set to: 0
--- Start: Offloading ---
Running on initial device? 0
Team 2 Thread: 0
Team 2 Thread: 1
Team 3 Thread: 0
Team 3 Thread: 1
Team 1 Thread: 0
Team 1 Thread: 1
Number of teams: 4
Team 0 Thread: 0
Team 0 Thread: 1
--- Stop: Offloading ---
Back on host: YES (HOST)
b24n0003@a1721:~/test$
```

# Lab 3: GPU Programming using OpenMP

C

Compile with:

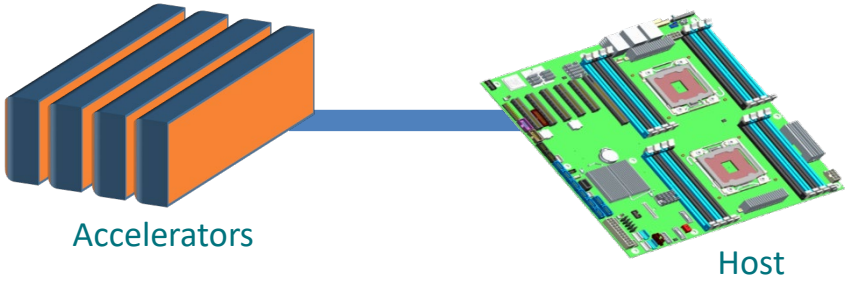
```
nvc -fast -mp=gpu -Minfo=mp,opt
nvc -fast -mp=gpu –gpu=mem:managed -Minfo=mp,opt
```

Fortran

Compile with:

```
nvfortran -fast -mp=gpu -Minfo=mp,opt
nvfortran -fast -mp=gpu –gpu=mem:managed -Minfo=mp,opt
```

# Optimizing Data Transfers is Key to Performance




- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects
  - Bandwidth host memory: hundreds of GB/sec
  - Bandwidth accelerator memory: TB/sec
  - PCIe Gen 4 bandwidth (16x): tens of GB/sec

- Unnecessary data transfers must be avoided, by
  - only transferring what is actually needed for the computation, and
  - making the lifetime of the data on the target device as long as possible.

# Role of the Presence Check

- If map clauses are not added to target constructs, presence checks determine if data is already available in the device data environment:

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    integer :: n, i
    real(kind=real32) :: a
    real(kind=real32), dimension(n) :: x
    real(kind=real32), dimension(n) :: y

!$omp target
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do     “present?(y)” “present?(x)”
!$omp end target
end subroutine
```

- OpenMP maintains a mapping table that records what memory pointers have been mapped.
- That table also maintains the translation between host memory and device memory.
- Constructs with no `map` clause for a data item then determine if data has been mapped and if not, a `map(tofrom:…)` is added for that data item.

# Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - Use `map` clauses to enforce direction of data transfer.
  - Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
subroutine caller
    ! Declarations omitted

!$omp target data map(to:x) &
                  map(tofrom:y)
    call saxpy(a, x, y, n)
!$omp end target
end subroutine
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted

!$omp target "present?(y)" "present?(x)"
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

# Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - Use `map` clauses to enforce direction of data transfer.
  - Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
void example() {
    float tmp[N], data_in[N], float data_out[N];
#pragma omp target data map(alloc:tmp[:N]) \
                        map(to:a[:N],b[:N]) \
                        map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
}   }
```

```
void zeros(float* a, int n) {
#pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}
```

```
void saxpy(float a, float* x, float* y, int n) {
#pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

# `target data` Construct Syntax

- Create scoped data environment and transfer data from the host to the device and back
- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],…]
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],…]
structured-block
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom | release | delete}:] list)
if(scalar-expr)
```

# `target enter and exit data` Construct Syntax

## Unstructured Data Directives C

```
void initialize(…) {
  //initialize data on the host
  array[i]=…;
  //initialize data on the GPU
  #pragma omp target enter data map(to: array[0:size]))
}
```

```
void deallocate(…) {
  //deallocate data on the GPU
  #pragma omp target exit data map(delete: array[0:size]))
  //deallocate data on the host
  free(array);
}
```

## Unstructured Data Directives Fortran

```
subroutine initialize(...)
  !initialize data on the host
  array(i)=…;
  !initialize data on the GPU
  !$omp target enter data map(to: array)
end subroutine initialize
```

```
subroutine deallocate(...)
  !deallocate data on the GPU
  !$omp target exit data map(delete: array)
  !deallocate data on the host
  deallocate (array)
end subroutine deallocate
```

# `target update` Construct Syntax

- Issue data transfers to or from existing data device environment
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

# Example: target data and target update

```
#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)
  }
```

# Lab 4: Data Management

C

Compile with:

```
nvc -fast -mp=gpu -Minfo=mp,opt
```

Fortran

Compile with:

```
nvfortran -fast -mp=gpu -Minfo=mp,opt
```

# OpenMP Resources

- OpenMP Webpage https://www.openmp.org/
- Specification https://www.openmp.org/specifications/
- OpenMP Books https://www.openmp.org/resources/openmp-books/
- IWOMP Conference https://www.iwomp.org/
  - IWOMP 2026 will be held on 7 – 9 Oct 2026 in conjunction with EuroMPI and the MPI Forum meetings in Vienna.
- OpenMP Reference Guide (Cheat Sheet) https://www.openmp.org/resources/refguides/