General purpose calculations on

HETEROGENEOUS COMPUTER SYSTEM (GPUs)

N. Shukla

High-Performance Computing Department CINECA

Casalecchio di Reno Bologna, Italy

CINECA

Email: n.shukla@cineca.it

www.cineca.it || Bologna 2022

Acknowledgments

This lectures slides are inspired and adopted from various sources:

- Tom Deakin (University of Bristol)
- Michael Klemm (AMD)
- Jeff Larkin (NVIDIA)
- Swaroop Pophale (ORNL, US)
- OpenMP GPU Offload Basics (Intel)
- and many others
- OpenMP 5.0.1 specification and examples https://www.openmp.org/resources/

Topics we will cover?

OpenMP execution model on CPUs

- Recap of OpenMP Worksharing

Introduction to OpenMP offload

- Host-device model
- How to offloads to the GPU?
- Compiler Support

Managing data movement

- Controlling **data transfer** between host and Device

Expressing parallelism

- Creating **parallelism** on the target device

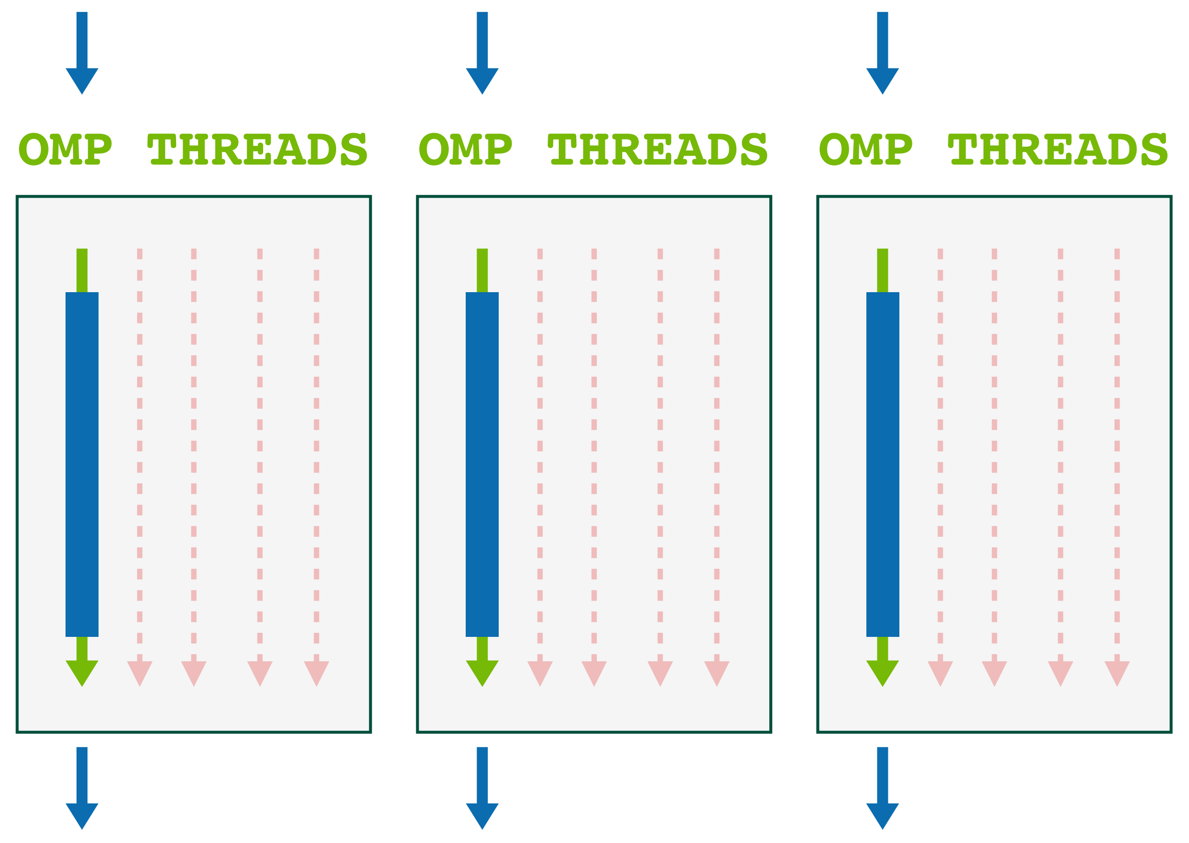
Best practise for OpenMP offloading on GPUs

Check points

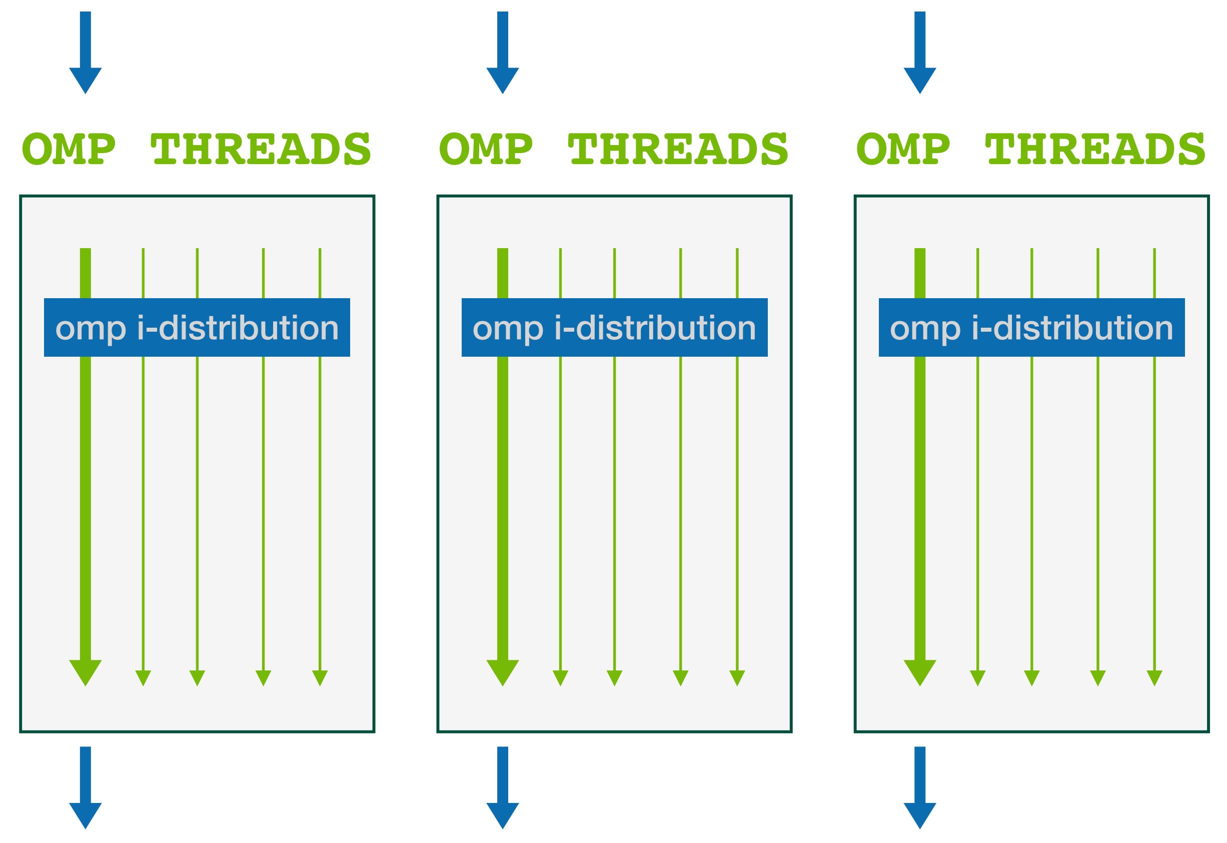
Revisit: OpenMP Worksharing

- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS

#pragma omp parallel



#pragma omp parallel for



Revisit: OpenMP Worksharing

Serial

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

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

Parallel

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

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

Parallel worksharing

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

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

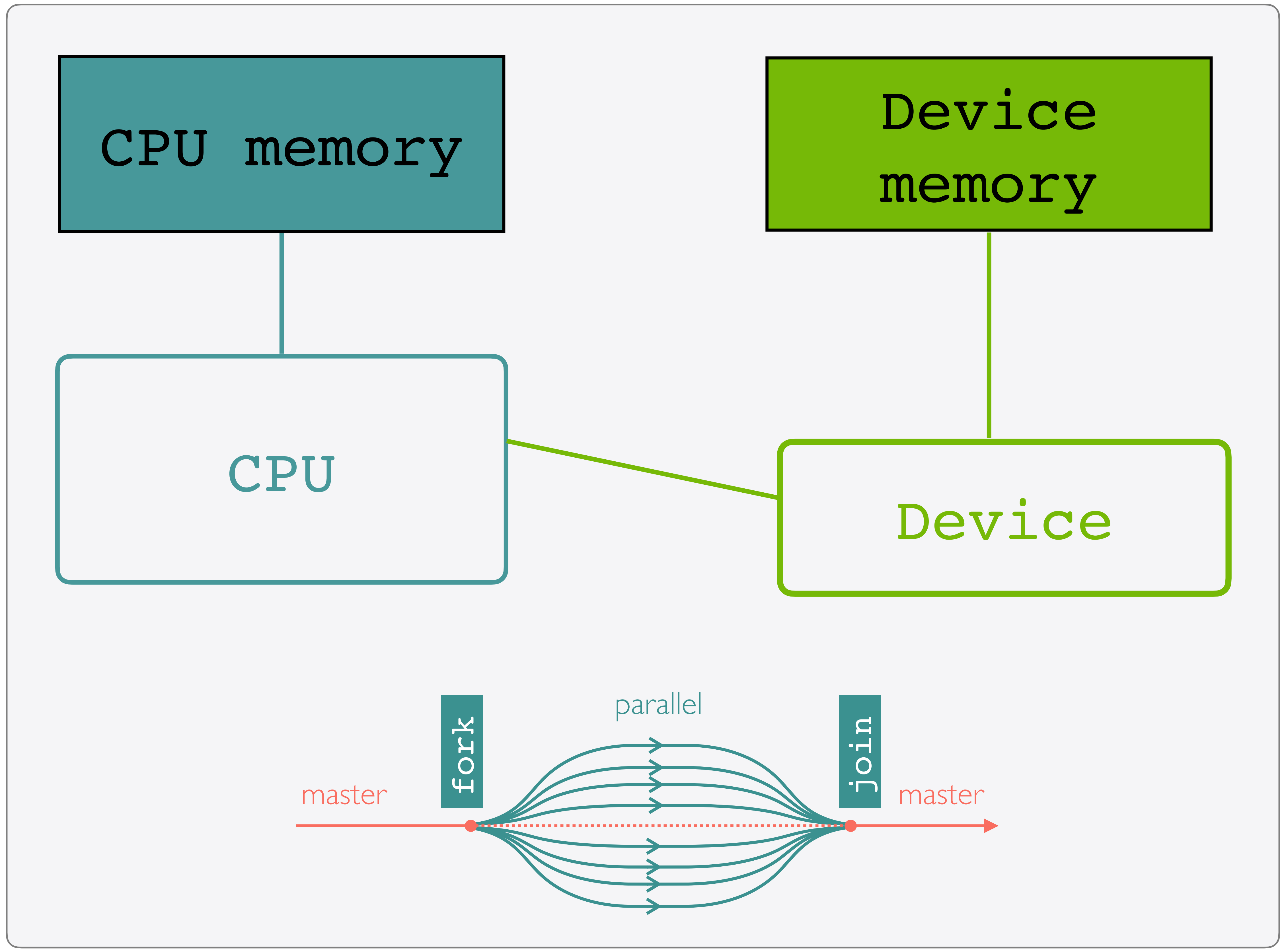
OpenMP uses host-device model

Device model

- Host-centric model
- One host device and multiple devices of the same type
- Devices are connected to host CPU via interconnect such as PCIe or NVLink
- Host and device have separate memory spaces

OpenMP Offload steps

- Identification of compute kernels
- Expressing parallelism within the kernels
- Managing data transfer between host to device



How to Offload?

- Program stats it execution on the host
- The target construct offloads the enclosed code to the accelerator
- When a target region is encountered, the code region is mapped and executed on the device
- By default, the code inside the target region executes sequentially
- At the end of the target region, the host thread waits for the target
  region code to finish, and continues executing the next statements

C/C++ API

*#pragma omp **target** [clause […] …]*

structured block

Fortran API

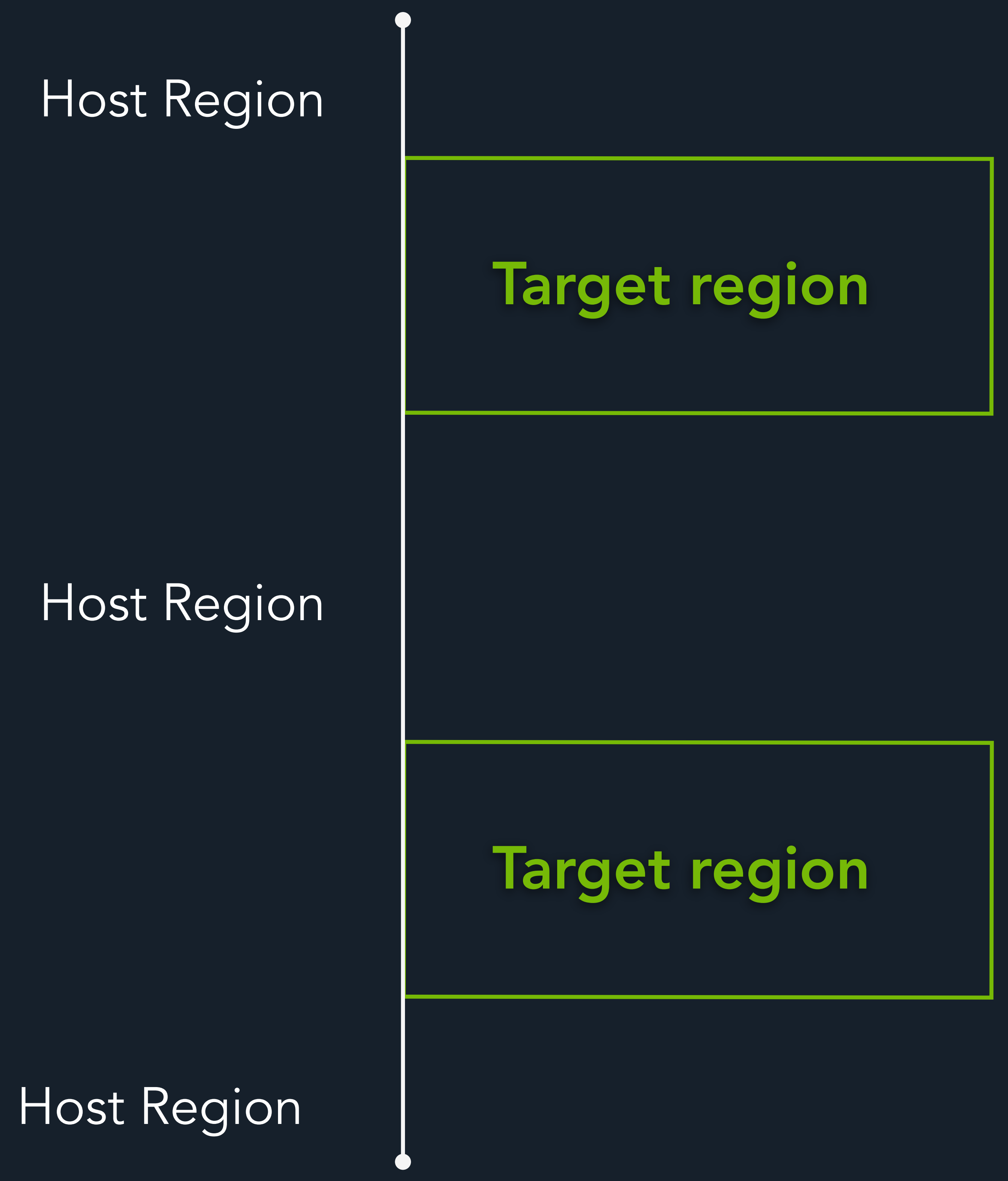
*!omp **target** [clause […] …]*

loosely/tightly-structured block

*!omp end **target** [clause […] …]*

Target clause gets kennel code running on the device

```
int main()
{
   #pragma omp target
   {
     // Line of codes

   }

   #pragma omp target
   {
     // Line of codes
   }

}
```



Running example for this presentation: DAXPY

```
// Adding OpenMP pragmas to parallelize work on the GPU

     int main(){
     /* initialise arrays */
      double tstart = omp_get_wtime();
       #pragma omp target
       {
           for ( i=0; i<n; I++)
              D[i] = A*X[i] + Y[I];

       }
      double tend = omp_get_wtime();
      printf("Time taken (s)= %.6f\n", tend-tstart );
     }
```

Timing code

Execution on target device

Timing code

Device runtime support

Runtime routines

- void omp_set_default_device (int dev_num)
- int omp_get_default_device(void)
- int omp_get_num_devices(void)
- int omp_get_num_teams(void)
- int omp_get_team_num(void)
- int omp_is_initial_device(void)
- int omp_get_initial_device(void)

Environment variables

- Control default device through

  OMP_DEFAULT_DEVICE

- Control offloading behaviour

  OMP_TARGET_OFFLOAD

Clause allowed on the target device

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

NVIDIA HPC COMPILER

Using OpenMP (target directive support since HPC SDK 21)

- OpenMP

-mp	Enable OpenMP targeting device
-mp=gpu	Enable OpenMP targeting device

- GPU Options

-gpu=ccXX	Set GPU target, specialise for one generation or many

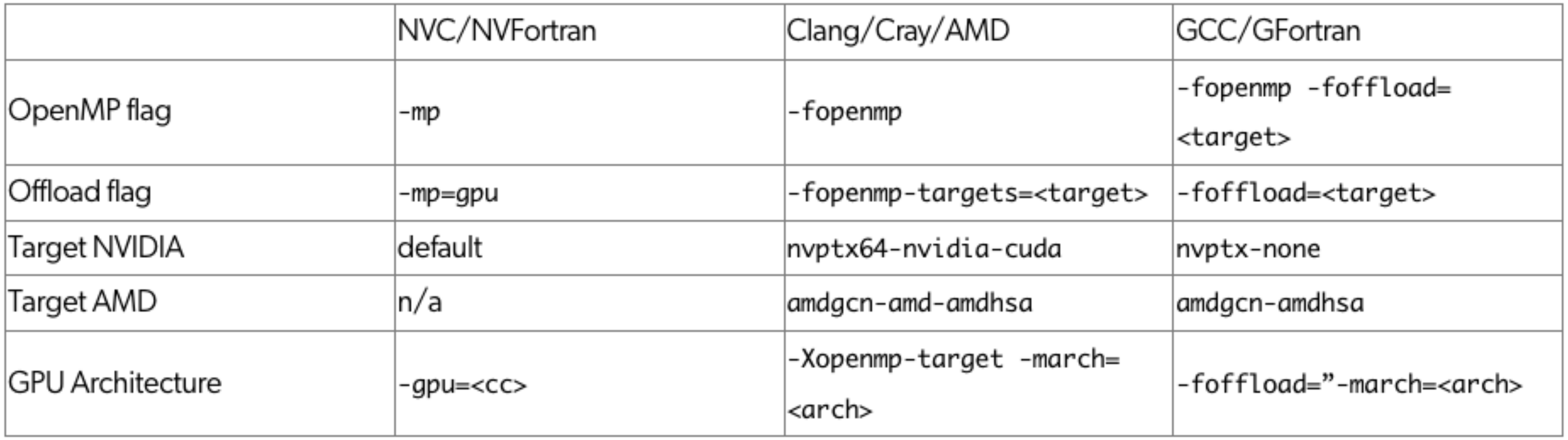
- Compiler Diagnostics

-Minfo=mp	Compiler diagnostics for OpenMP

- Environment variable for NOTIFY
  - export NVCOMPILER_ACC_NOTIFY = 1|2|3

```
$ nccx -mp=gpu -gpu=managed -Minfo=mp -o binary cOmpOffload.c
```

Compiler support



	NVC/NVFortran	Clang/Cray/AMD	GCC/GFortran
OpenMP flag	`-mp`	`-fopenmp`	`-fopenmp -foffload=<target>`
Offload flag	`-mp=gpu`	`-fopenmp-targets=<target>`	`-foffload=<target>`
Target NVIDIA	default	`nvptx64-nvidia-cuda`	`nvptx-none`
Target AMD	n/a	`amdgcn-amd-amdhsa`	`amdgcn-amdhsa`
GPU Architecture	`-gpu=<cc>`	`-Xopenmp-target -march=<arch>`	`-foffload=”-march=<arch>`

Host and device data

Host and device have separate memory spaces

- Data needs to mapped to the device
- Mapped data can not by accessed by the host during execution

Default behaviour

- Scalars are mapped firstprivate (i.e do not get copied back to the host)
- Statically allocated arrays are mapped tofrom
- Heap arrays are NOT mapped by default
- Data allocated on the heap needs to be explicitly copied to/from the device

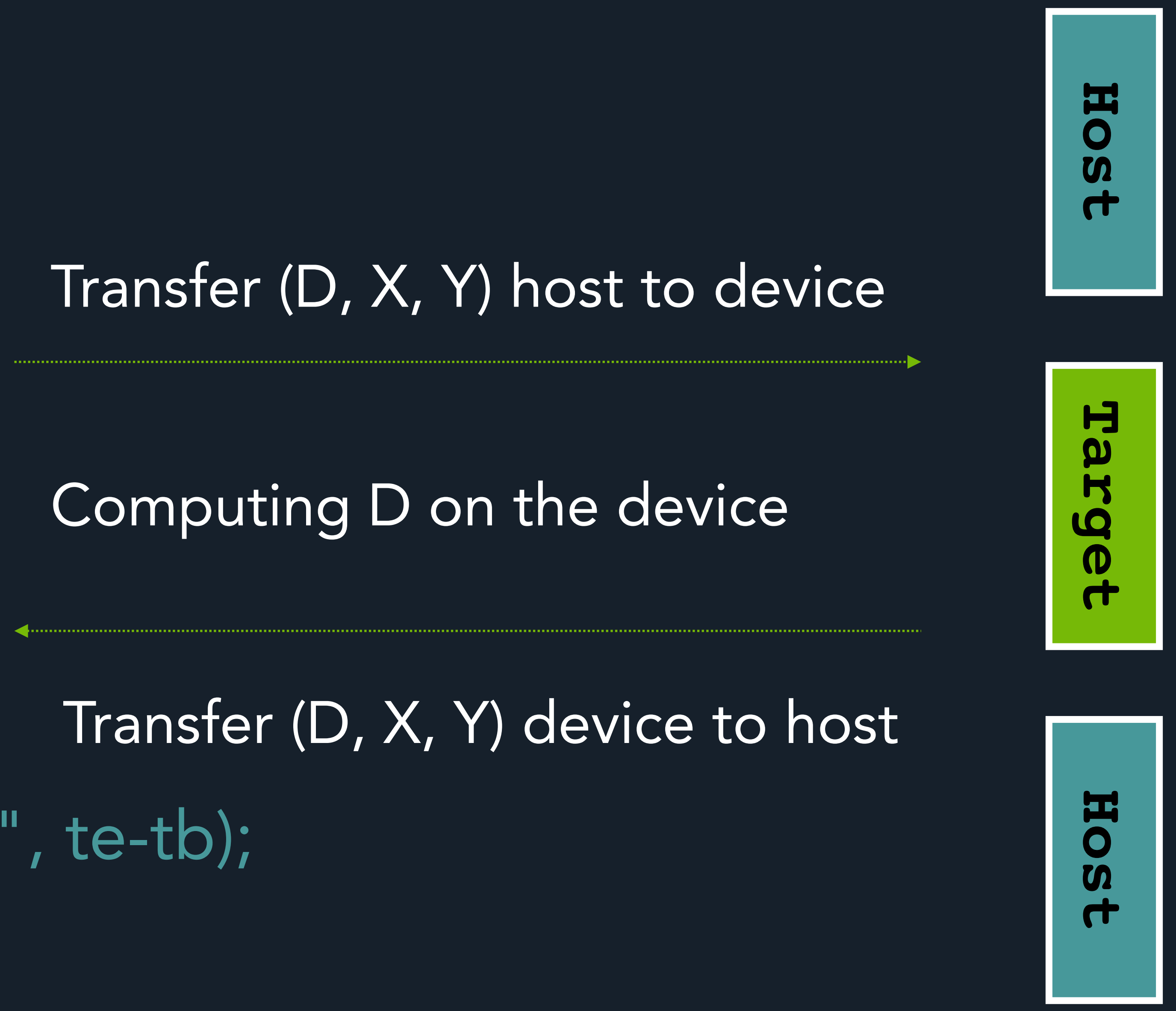
OpenACC is the default way that data is handled when entering a parallel work region.

Implicit mapping rules on target

Default behaviour

- Scalars and statically allocated arrays that are referenced in the target region are moved onto the device implicitly before execution
- Only the statically allocated arrays are moved back to the host after the target region completes

```
void daxpygpu() {
        double A, D[n], X[n], Y[n];
        int A = 16.0;
        double tb, te;
        tb = omp_get_wtime();
        #pragma omp target {
        for (int i = 0; i < n; i++)
                D[i] = A*X[i] + Y[I]; }
       tb = omp_get_wtime();
       printf("Time of kernel: %lf\n", te-tb);
}
```


Transfer (D, X, Y) host to device
Computing D on the device
Transfer (D, X, Y) device to host
Host
Target
Host

Managing data movement

Data used to the region may be implicitly or explicitly mapped to device

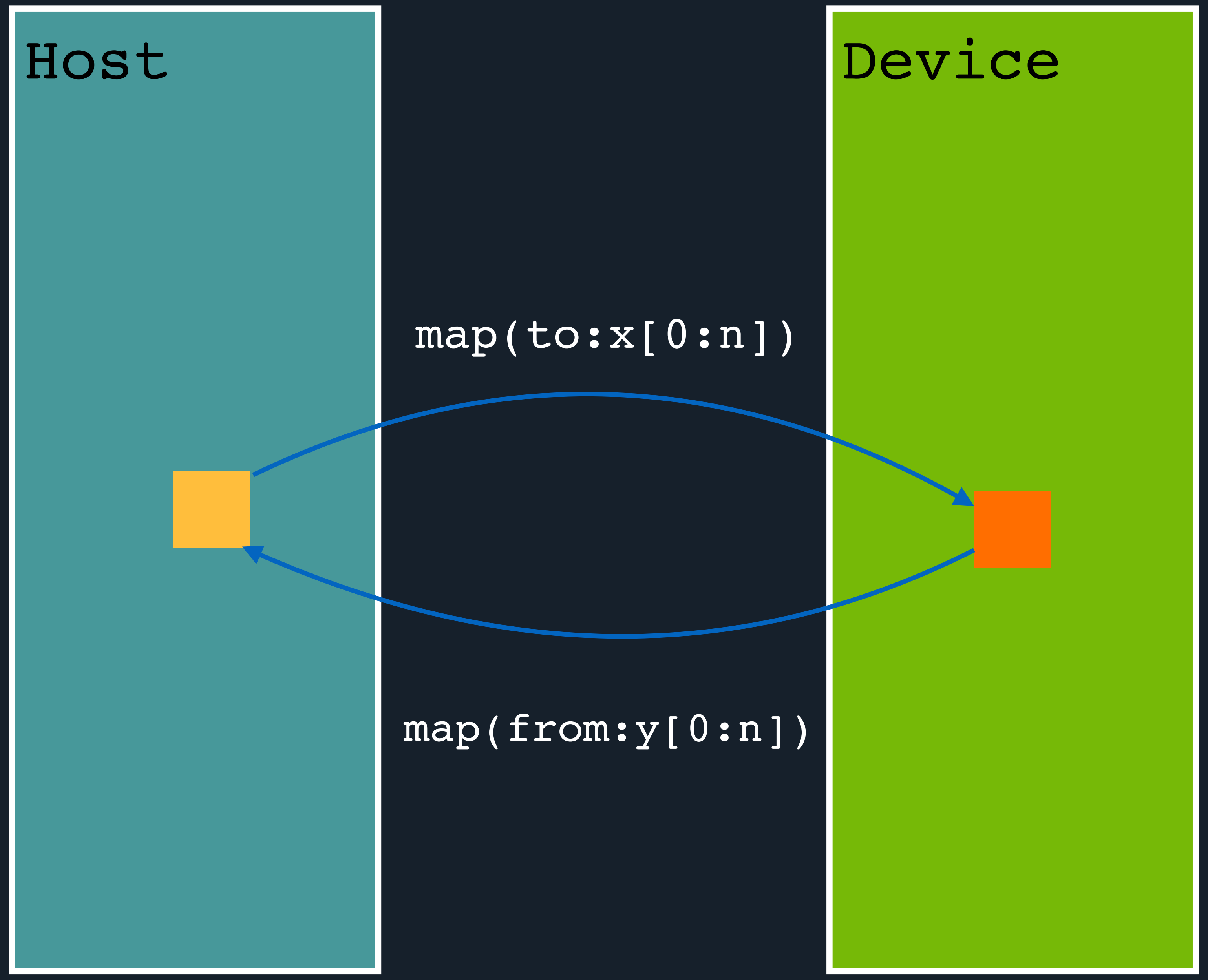
"The map clause specifies how an original list item is mapped from the current task's data environment to a corresponding list item in the device data environment of the device identified by the construct."

- OpenMP provides more control via the map clause on the target construct
- Specify the transfer of data between host and device on a target region

```
#pragma omp target map()
```

where list is a list of variables and map-type is one of

- **to** copy the data to the device on entry
- **from** copy the data to the device on entry
- **tofrom** copy the data to the device on entry and back on exit
- **alloc** allocate an uninitialised copy on the device (don't copyvalues)



OpenMP offload: example using omp target

/* C code to offload DAXPY to device using static arrays */

```
void daxpygpu()
{
   double A, X[n], Y[n];
   double tb, te;
   tb = omp_get_wtime();
   #pragma omp target map(to:X,Y) map(from:D)
   for (int i = 0; i < n; i++){
           D[i] = A*X[i] + Y[i];
   }
   tb = omp_get_wtime();
   printf("Time of kernel: %lf\n", te-tb);
}
```

/* C code to offload DAXPY to device with map clause using dynamics arrays */

```
void daxpygpu(double *D, double *X, double *Y, size_t n)
{
   int A = 16.0;
   double tb, te;
   tb = omp_get_wtime();
#pragma omp target map(to:X[0:n], Y[0:n]) map(from:D[0:n])
   for (i = 0; i < n; i++){
           D[i] = A*X[i] + Y[i];
    }
   tb = omp_get_wtime();
   printf("Time of kernel: %lf\n", te-tb);
}
```

Offloading Multiple kernels

/*C code for multiple offload kernels */

```
  …
#pragma omp target map(to: A, B) map(from: C)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
  }

/*
Some computation using C (no changes to A, B or C)
*/

#pragma omp target map(to: A, B, C) map(from: D)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[i][j];
      }
    }
  }
…
…
```

Is this optimal ?

NO

A and B are unchanged between the two target regions

Keeping data on the device

- Moving data between the host and device is expensive on a lot of current hardware
- Avoid mapping data in every target region if it can be kept on the device between target regions
- Use target enter data and target exits data constructs to control device data environment
- **target data** constructs just map data and do not offload any code
- target **update** construct copies values between host and device target constructs

Keeping data on the device

/*C code for multiple offload kernels with structured data mapping using target data map*/

```
  …
#pragma omp target data map(to: A, B)
{
#pragma omp target map(from: C)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }end-for
    }end-for
  } end target

/*
Some computation on host using C (no changes to A, B or C)
*/

#pragma omp target map(to: C) map(from: D)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[i][j];
      }
    }
  }
}//end target-data
…
…
```

Multiple offload kernels using target enter/exit data

/*C code for multiple offload kernels using target enter/exit data map*/

```
  …
#pragma omp target enter data map(to: A, B)

#pragma omp target map(from: C)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        C[i][j] = A[i][j] + B[i][j];
      }end-for
    }end-for
  } end target

/*
Some computation on host using C (no changes to A, B or C)
*/

#pragma omp target map(to: C)   {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[i][j];
      }
    }
  }

#pragma omp target exit data map(release: C) map(from: D)
```

- Use target enter data and target exit data constructs to control device data environment
- Bulk transfer happen at the beginning and end, not for every target region in the big loop
- Target regions inherit the existing data movement

Target update construct

```
!$omp target enter data map(to: A, B, C)
!$omp target
... ! Use A, B and C on device
!$omp end target

! Copy A from device to host
!$omp target update from(A(1:N))

! Change A on the host
A = 1.0

! Copy A from host to device
!$omp target update to(A(1:N))

!$omp target
... ! Use A, B and C on device
!$omp end target

!$omp target exit data map(from: C)
```

Often need to transfer data between host and device between different target regions.

E.g. the host does something between the two regions.

Use the update construct to move the data explicitly between host and device, in either direction.

Remember: direction is from the host's perspective.

Target update

/*C code for multiple offload kernels using target data map and target update*/

```
...
#pragma omp target data map(to: A, B) map(alloc: C, D) {

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

}

#pragma omp target update from(C)         //Updates C device →  host

/*Some computation using C on host (no changes to A, B or C)*/

#pragma omp target map(from: D)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[I][j];
    }
    }
 }
}//end target-data
```

```
...
#pragma omp target data map(to: A, B) map(alloc: C, D) {

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

}

#pragma omp target update from(C)         //Updates C device →  host

/*Some changes to A (no changes to B or C)*/

#pragma omp target update to(A)         //Updates A Host →  Device

#pragma omp target map(from: D)
  {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        D[i][j] = A[i][j] + B[i][j] C[I][j];
    }
    }
 }
}//end target-data
```

Asynchronous offloading

```
!$omp target nowait
!$omp teams distribute parallel do
do i = 1, 10000000
  ... ! Lots of work
end do
!$omp end teams distribute parallel do
!$omp end target
! Host just continues because of nowait

call expensive_io_routine()

! Wait for target task to finish
!$omp tastwait
```

- A host task is generated that encloses the target region
- The **nowait** clause indicates that the encountering thread does not wait for the target region to complete
- The host thread can continue working asynchronously with the device!

- Must synchronise using taskwait, or at a barrier (explicit or implicit) depending on host threading design.

Unified Shared Memory

Single address space over CPU and GPU memories

```
#pragma omp requires unified_shared_memory

// No data directive or mapping needed for pointers a, b, c
#pragma omp target teams distribute parallel for
  for (int i=0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
```

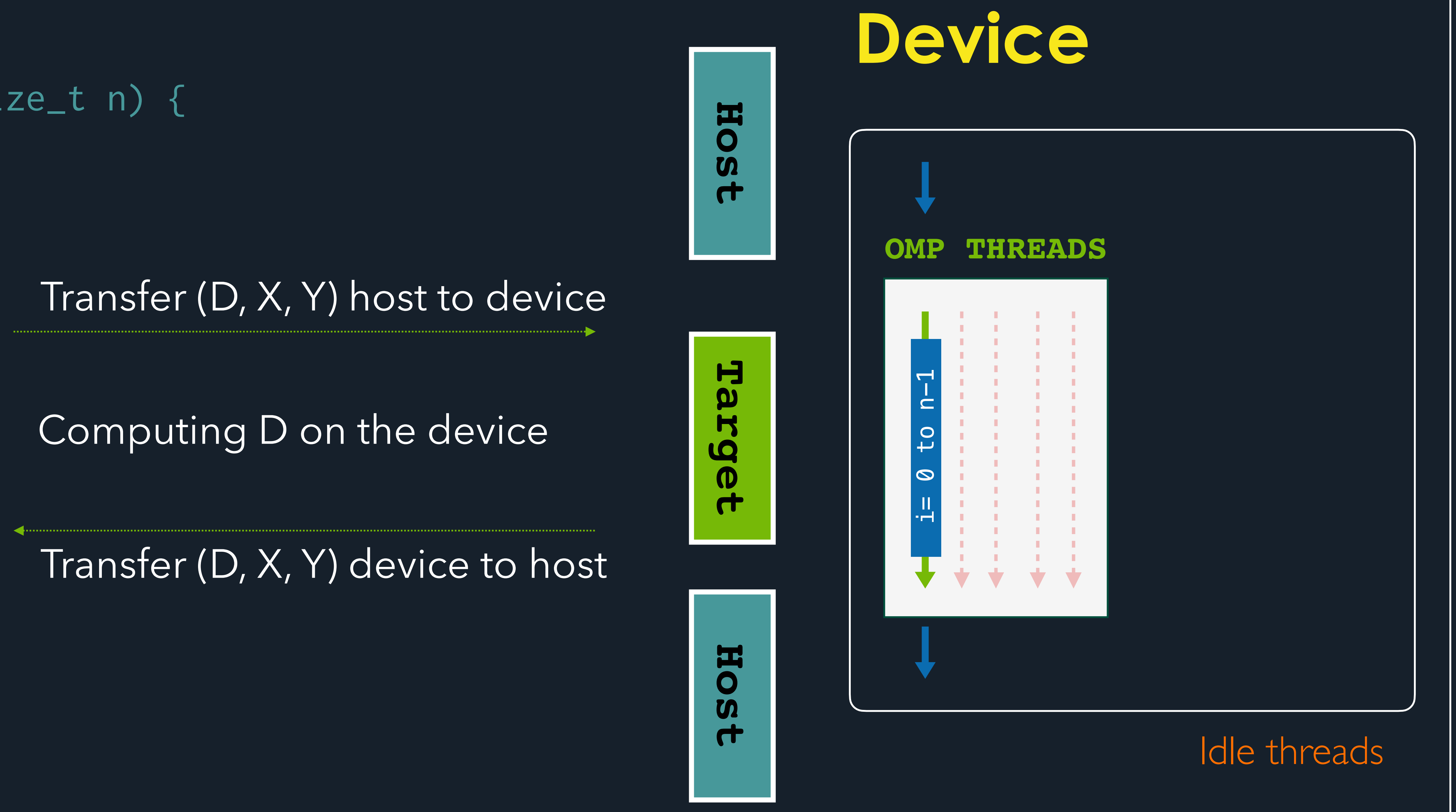
Warning: may not be supported by all compiler

Expressing parallelism

DAXPY: Dynamically allocated arrays

The target construct is a task generating construct

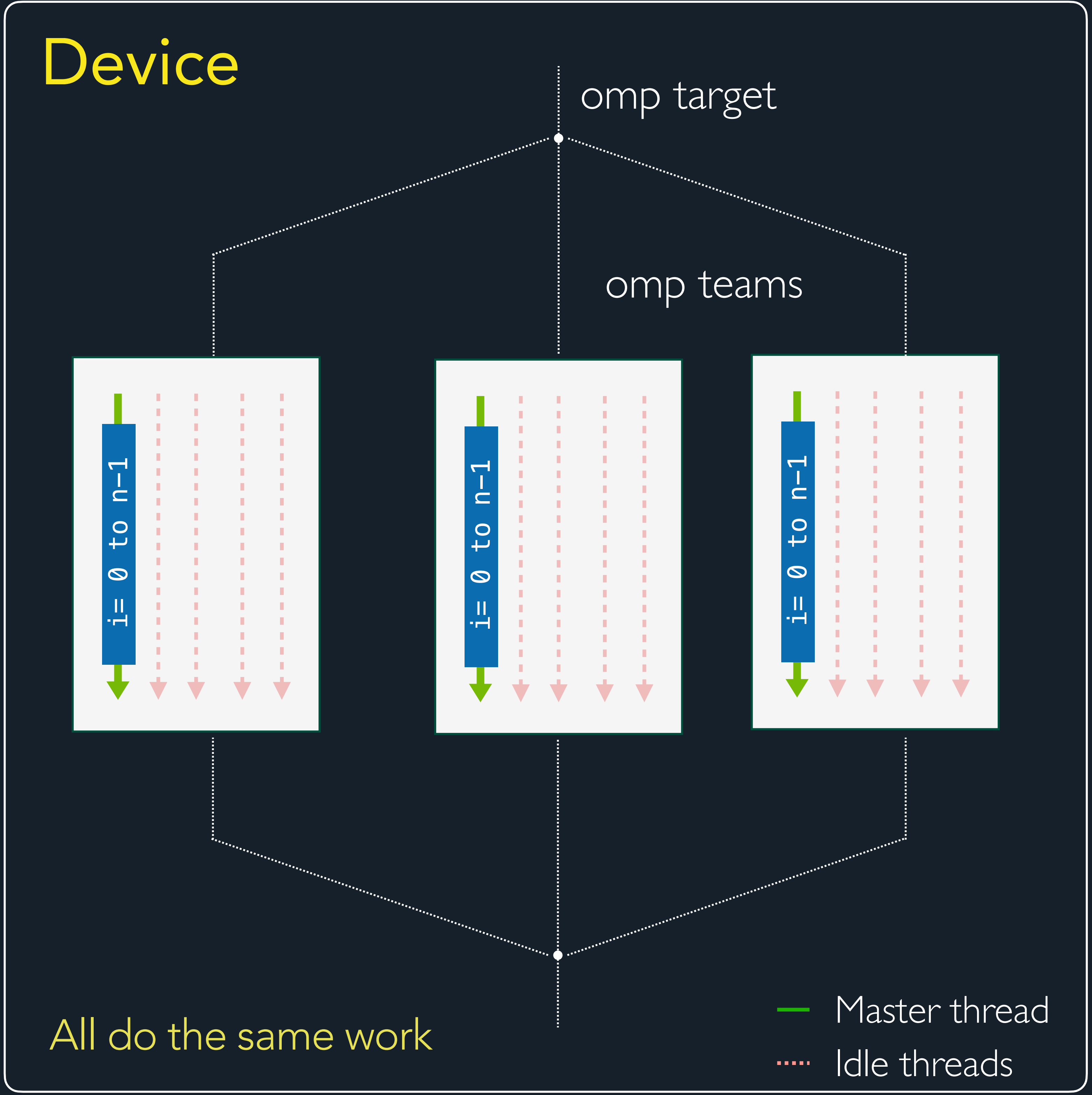
```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n) {
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:X[0:n], Y[0:n]) \
     map(tofrom:D[0:n])
    for (i = 0; i < n; i++){
           D[i] = A*X[i] + Y[i];
    }
    tb = omp_get_wtime();
    printf("Time of kernel: %lf\n", te-tb);
}
```


Transfer (D, X, Y) host to device
Computing D on the device
Transfer (D, X, Y) device to host
Host
Target
Host
Device
OMP THREADS
i= 0 to n-1
Idle threads

Teams constructs

The team construct creates a league of initial threads

- Each initial thread is a team of one thread
- Group of one or more threads are called Teams
- A set of thread teams called league
- Synchronisation does not apply across teams
- Execution continues on the master threads of each team (redundantly



Teams constructs

Support multi-level parallel device

Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],...]
```

Syntax (Fortran):

```
!$omp teams [clause[[,] clause],...]
```

Clauses

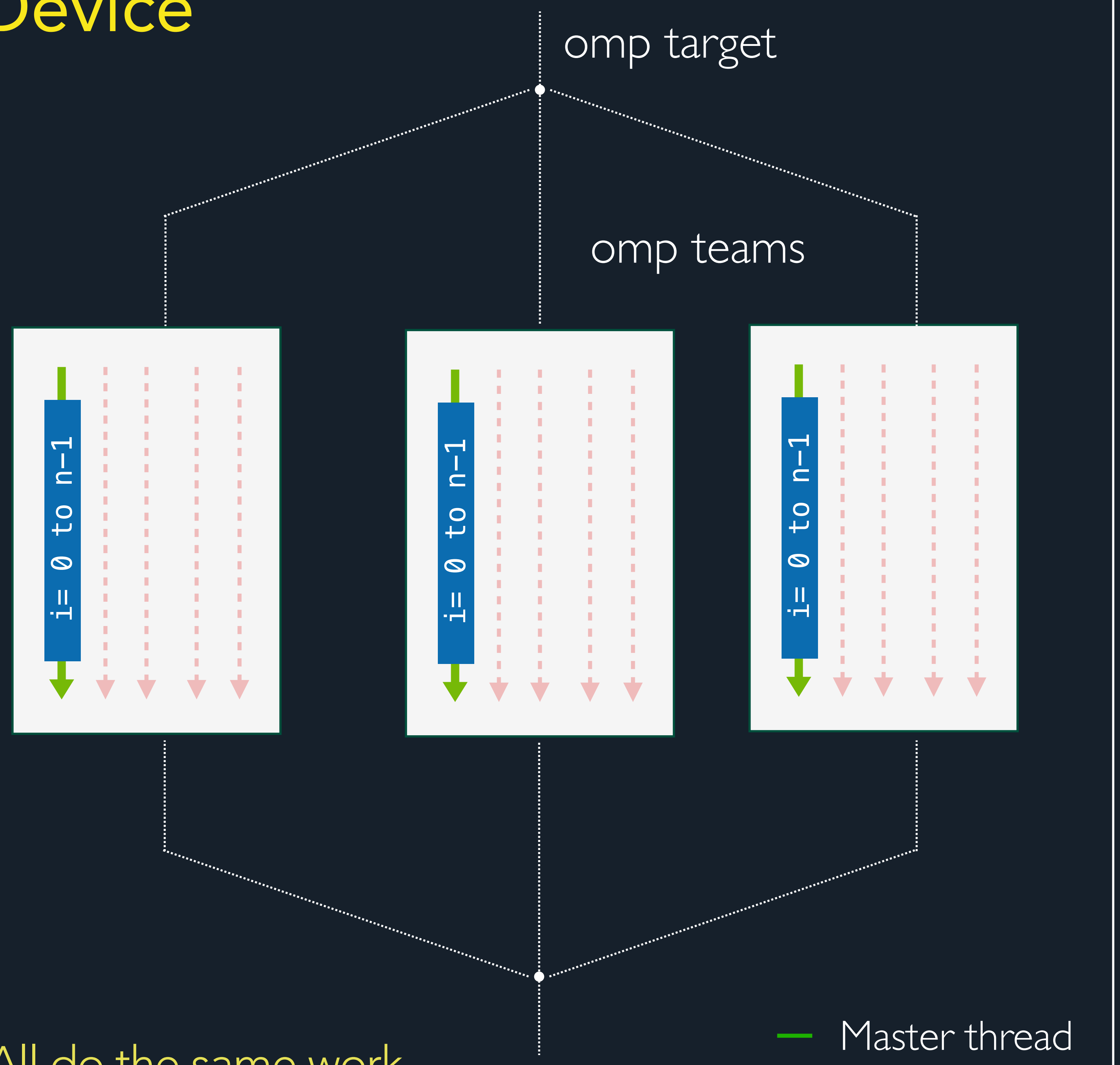
num_teams(*integer-expression*)

thread_limit(*integer-expression*)

default(shared | firstprivate | private none)
private(*list*),

firstprivate(*list*), shared(*list*), reduction(*operator:list*)

Device



All do the same work

— Master thread
····· Idle threads

Teams constructs creates league of teams

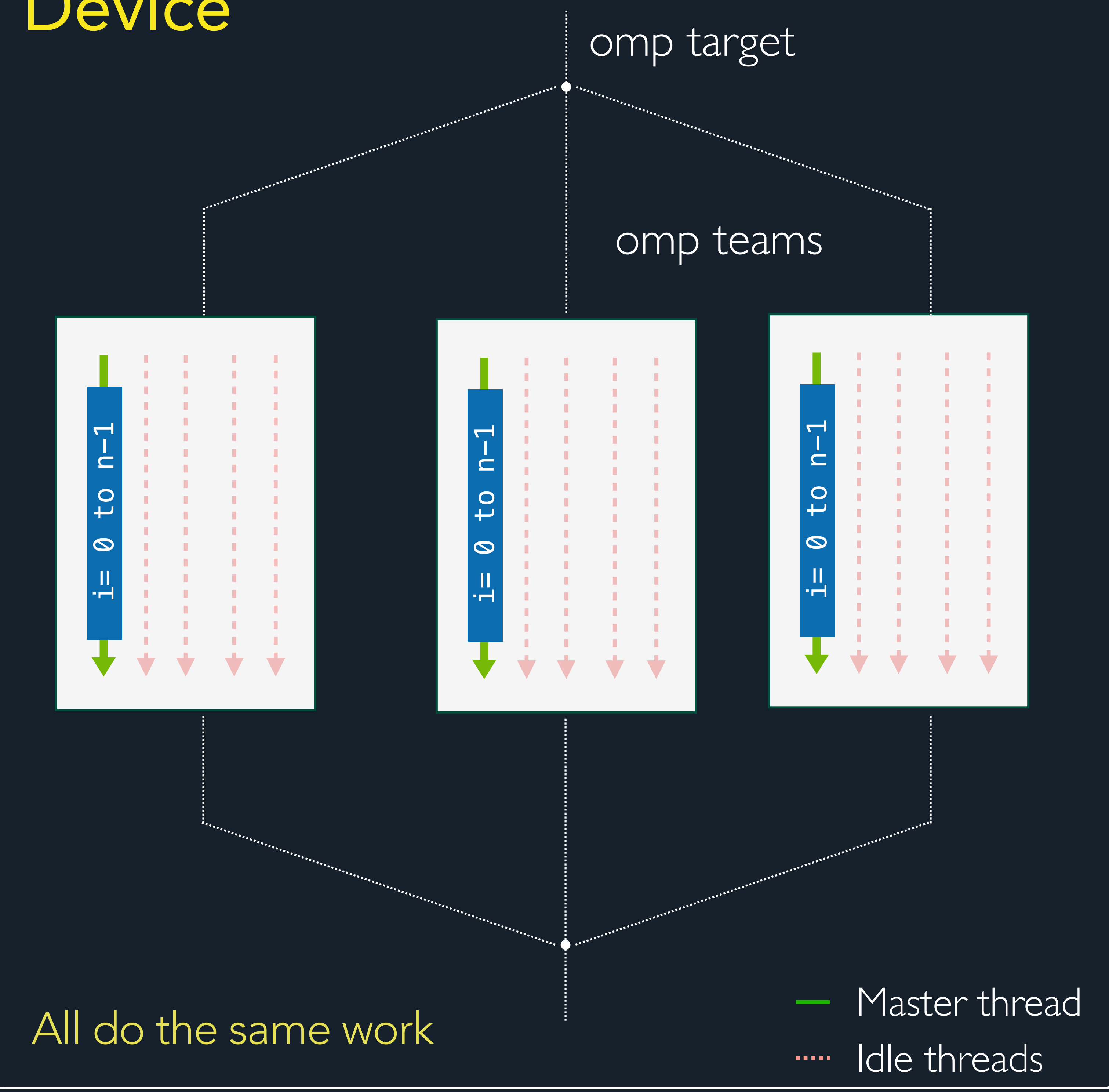
Initial thread of each team executes the code region

```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n)
{
  #pragma omp target map(to:X[0:n], Y[0:n]) map(tofrom:D[0:n])
 {
  #pragma omp teams num_teams(3)
  for (i = 0; i < n; i++)
         D[i] = A*X[i] + Y[i];
  }
}
```

Host

Target

Host



Distribute constructs shares works across the teams

The distribute construct divides loop iterations across the different teams

- Worksharing within a league
- Nested inside a teams region
- Can specify distribution schedule
- Similar to for/do construct for parallel regions

Syntax

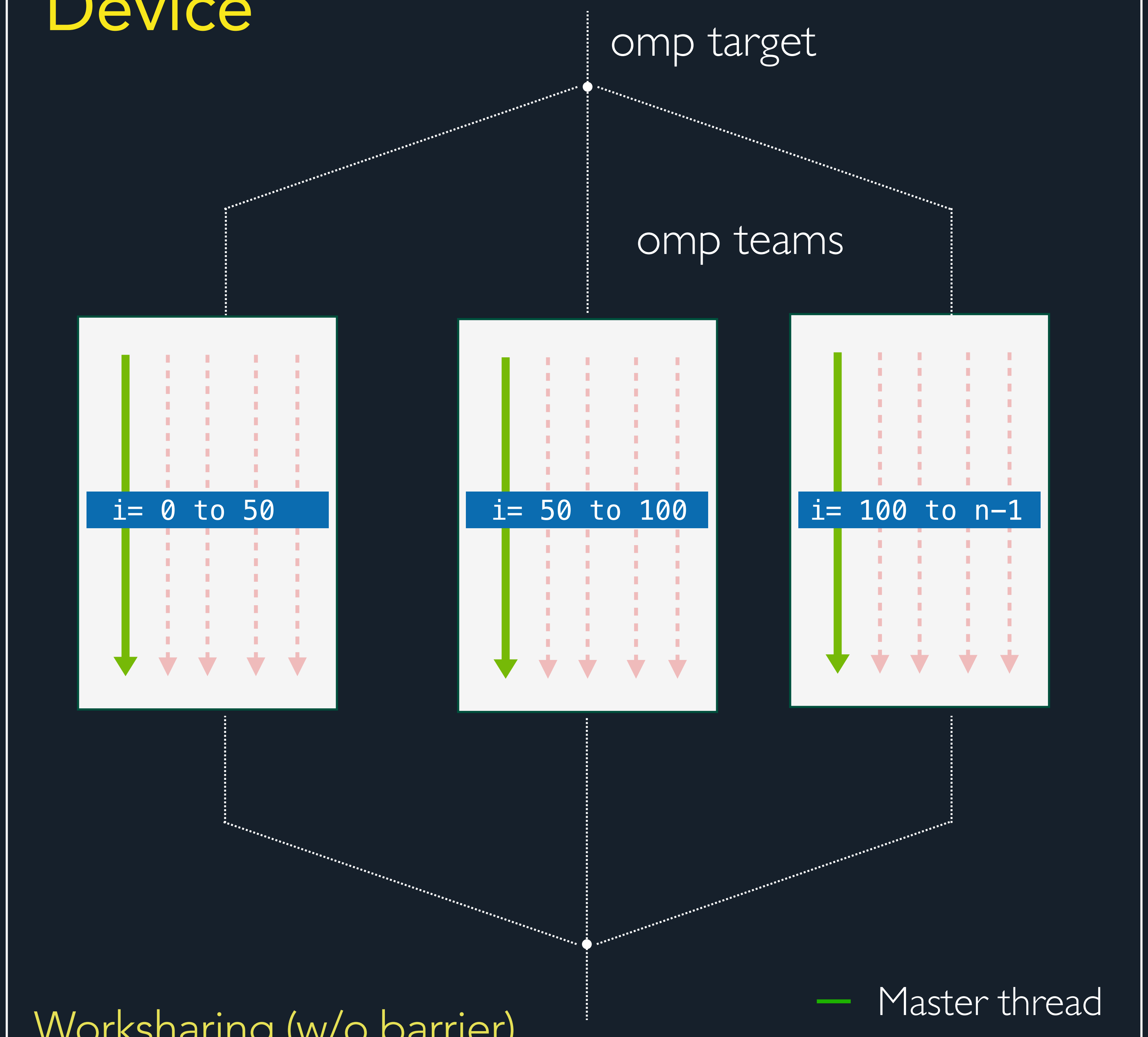
- C/C++:

  #pragma omp teams [clause[[,] clause],…]

- Fortran:

  !$omp teams [clause[[,] clause],…]

Device



Worksharing (w/o barrier)

— Master thread
····· Idle threads

Distribute constructs shares works across the teams

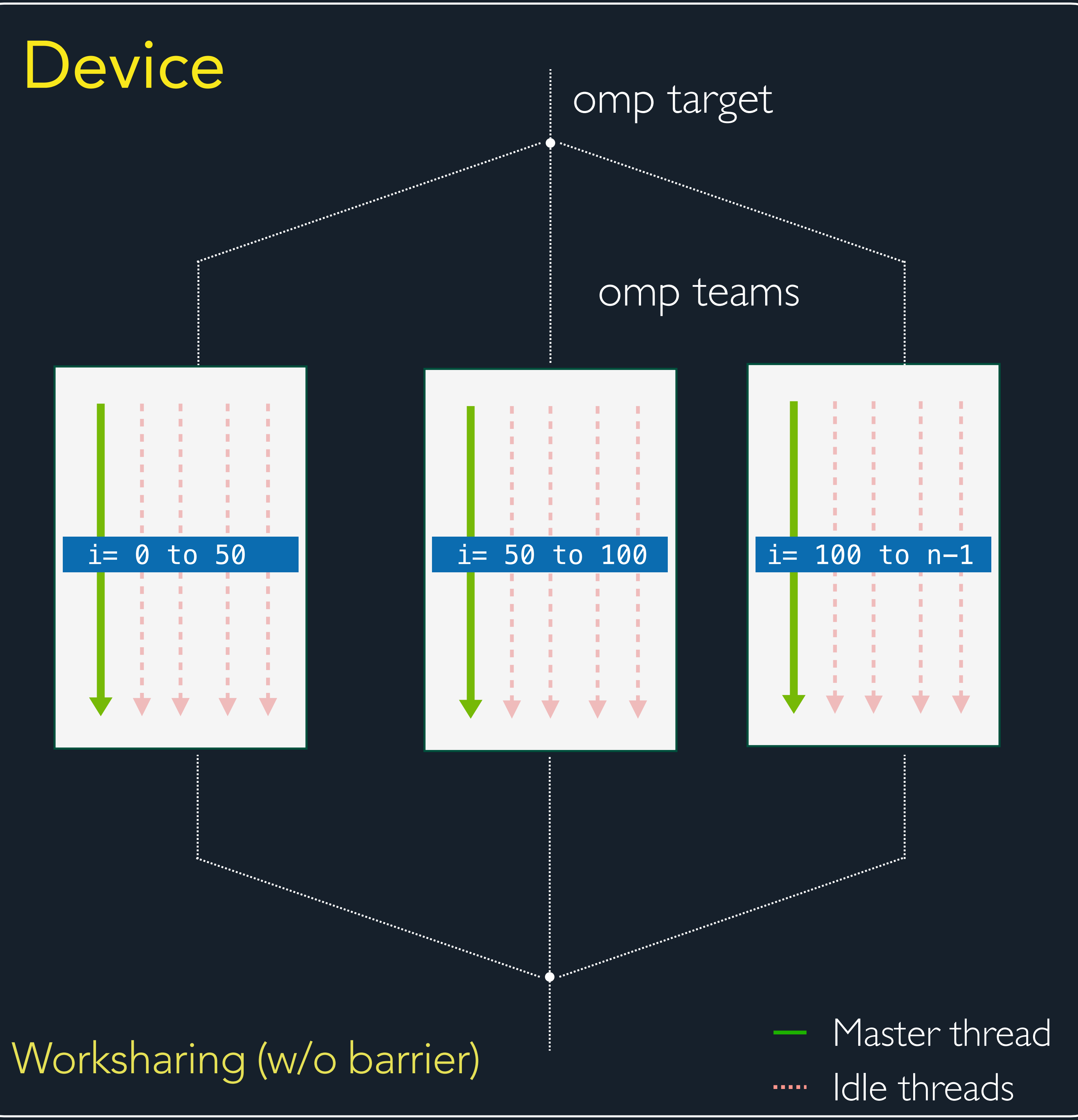
A league of thread teams is created, and loop iterations are distributed and executed by the initial teams

```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n)
{
  double tb, te;
  tb = omp_get_wtime();
  #pragma omp target map(to:X[0:n], Y[0:n]) map(tofrom:D[0:n])
 {
  #pragma omp teams num_teams(3) distribute
  for (i = 0; i < n; i++)
         D[i] = A*X[i] + Y[i];
  }
  tb = omp_get_wtime();
  printf("Time of kernel: %lf\n", te-tb);
}
```

Host

Target

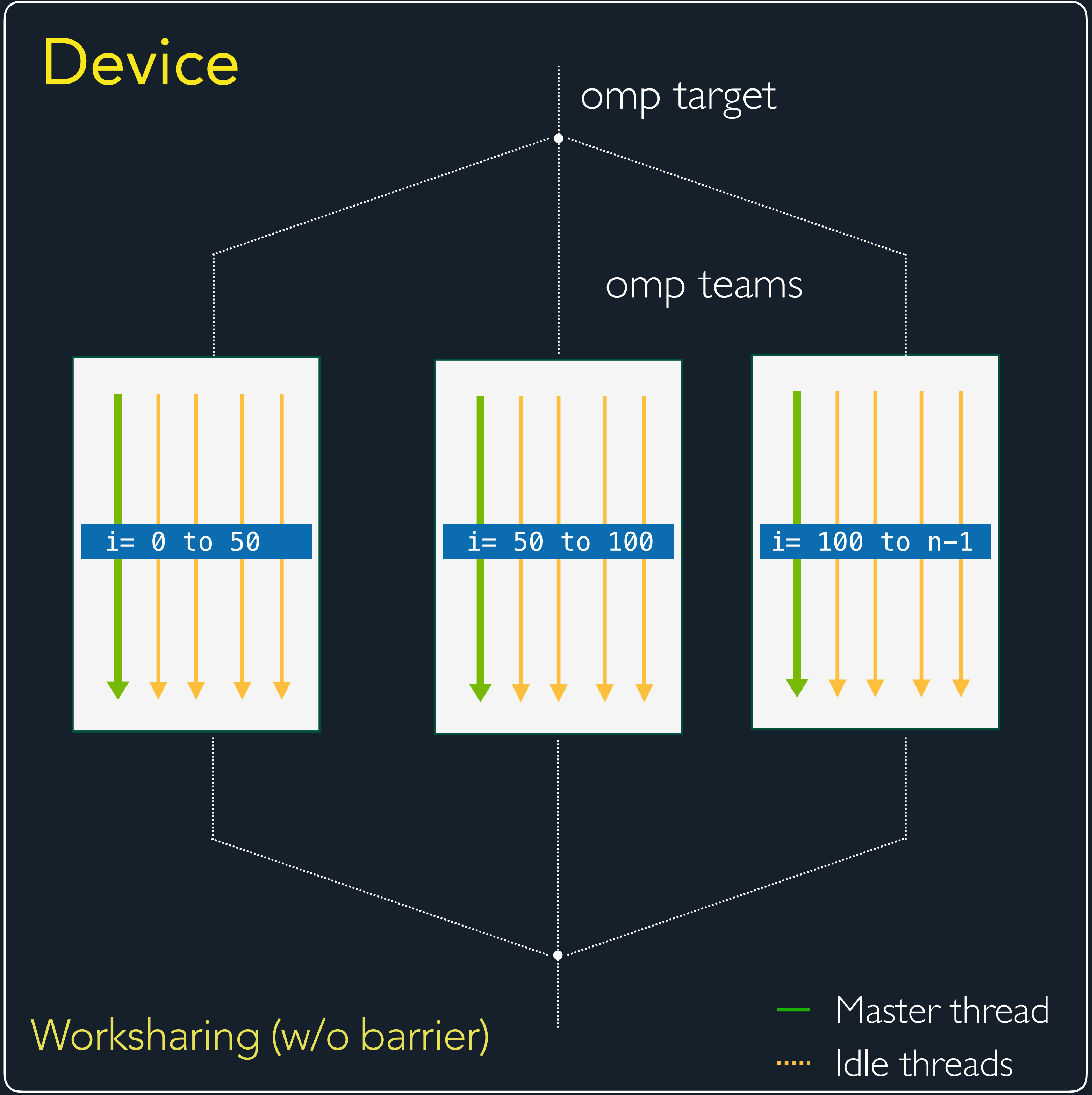
Host



Parallel for shares work to all threads of the teams

Same semantics on the CPU

- Launches threads within the team and the do distributes iteration across the threads in a team
- Note, iterations that were assigned to the team by the distribute construct are distributed across threads in the team
- Can use the schedule clause too
- Finally have lots of parallel execution!



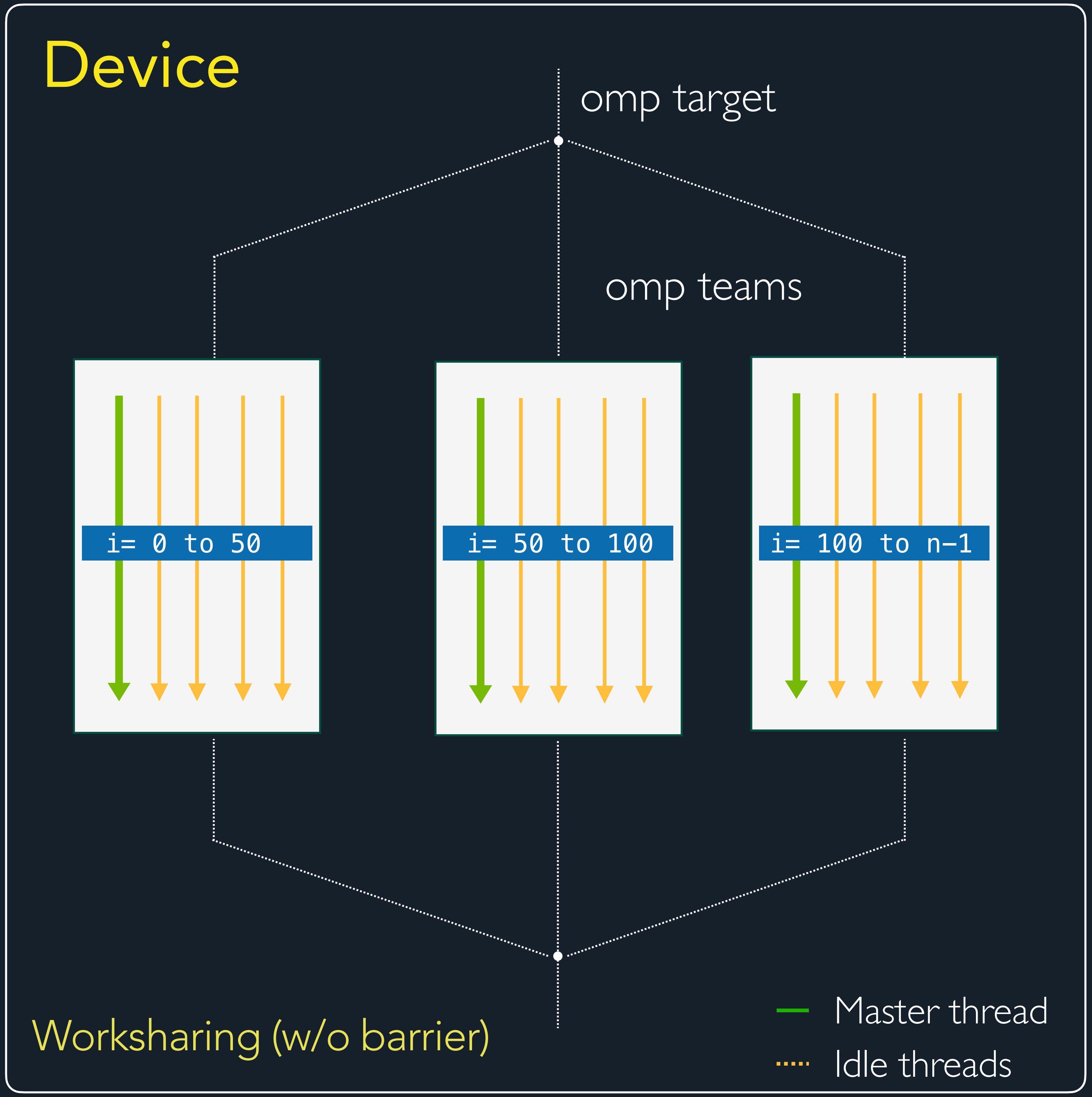
Parallel for shares work to all threads of the teams

```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n)
{
  double tb, te;
  tb = omp_get_wtime();
  #pragma omp target map(to:X[0:n], Y[0:n]) map(tofrom:D[0:n])
 {
  #pragma omp teams num_teams(3) distribute for
  for (i = 0; i < n; i++)
         D[i] = A*X[i] + Y[i];
  }
  tb = omp_get_wtime();
  printf("Time of kernel: %lf\n", te-tb);
}
```

Host

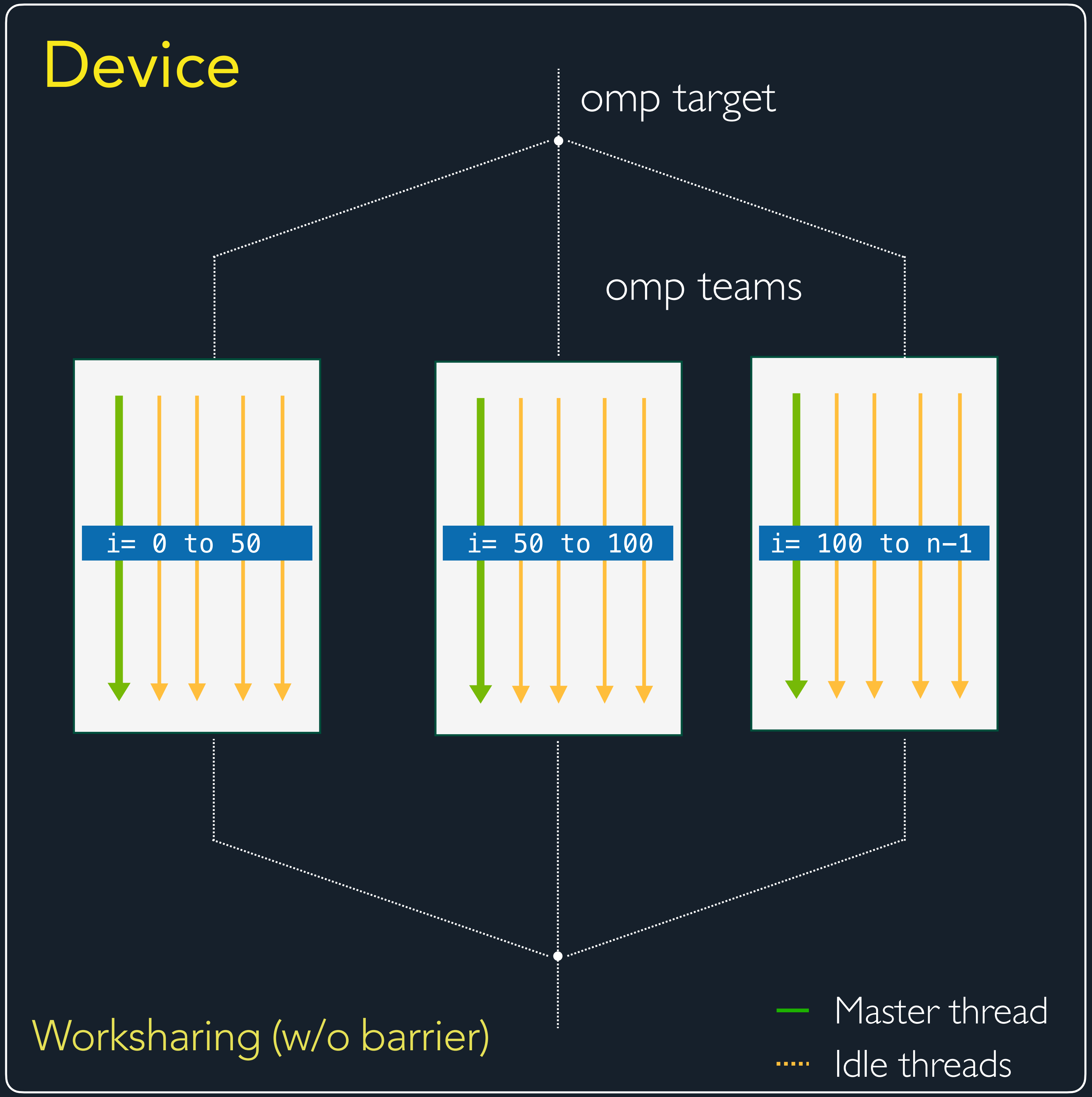
Target

Host



Expressing Parallelism: SIMD

- The SIMD construct is also valid on the distribute parallel do/for construct.
- OpenMP says this means SIMD instructions are generated
- Minor implementation details differ between compilers
- Compilers sometimes ignore different parts of the construct, depending on the situation
- !$omp target teams distribute parallel do is a portable solution for practically obtaining the same parallelism across compilers.



Multi-level parallel parallelism

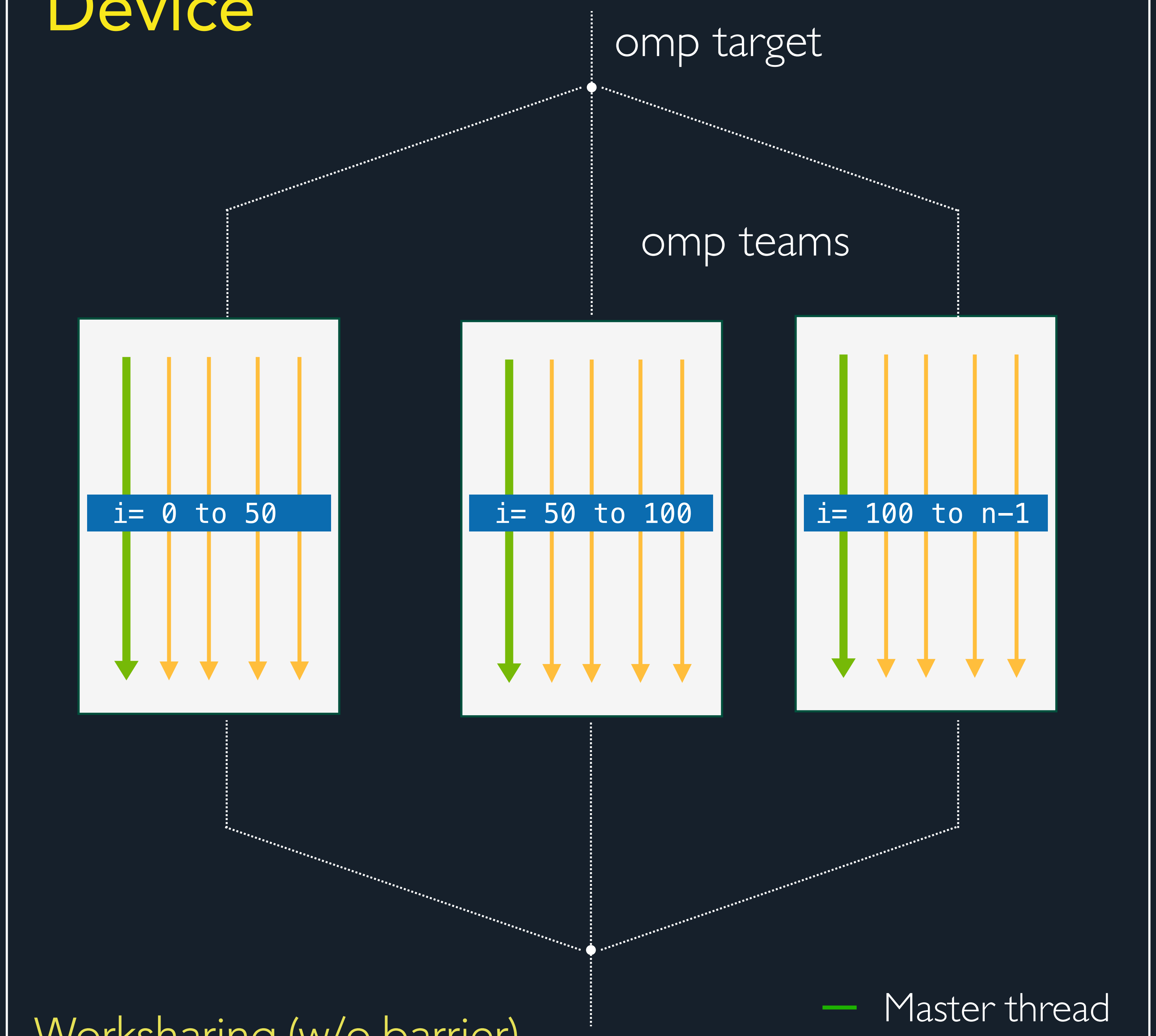
OpenMP defines composite constructs for convenience

```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n) {

  double tb, te;
  tb = omp_get_wtime();
#pragma omp target teams num_teams(3) distribute parallel for simd \
map(to:X[0:n], Y[0:n]) map(tofrom:D[0:n])
 for (i = 0; i < n; i++)
         D[i] = A*X[i] + Y[I];

  tb = omp_get_wtime();
  printf("Time of kernel: %lf\n", te-tb);
}
```

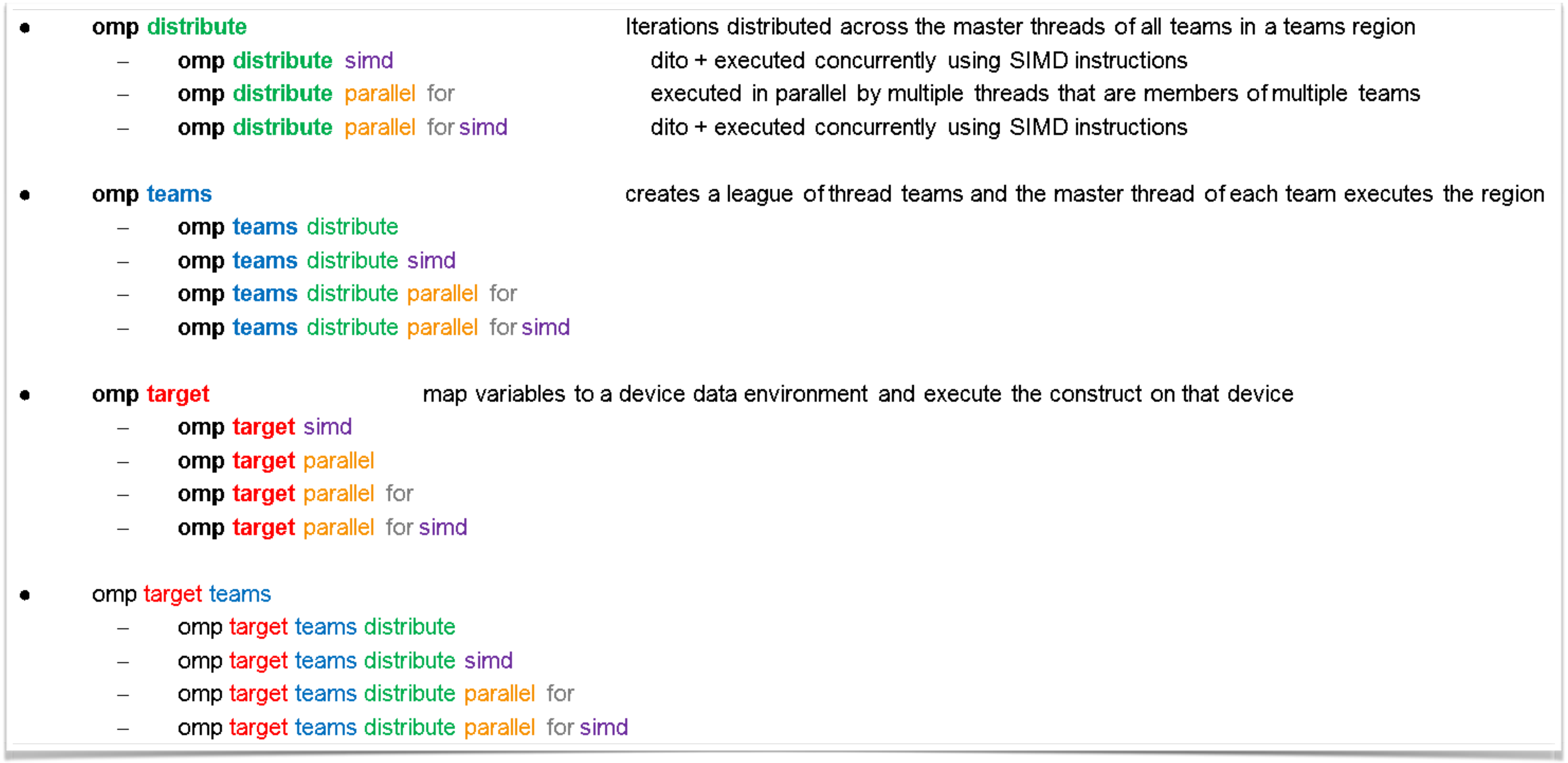
Host

Target

Host



Combined constructs



- 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

Best practise for OpenMP on GPUs

- Always use the teams and distribute directive to expose all available parallelism
- Aggressively collapse loops to increase available parallelism
- Use the target data directive and map clauses to reduce data movement between CPU and GPU
- Use accelerated libraries whenever possible
- Use OpenMP tasks to go asynchronous and better utilize the whole system
- Use host fallback (if clause) to generate host and device code

Let’s talk about performance

- Transferring data between host and device can be performance killer
- Transfers between host and device are relatively very slow
- Only transfer the data which is required, otherwise minimize it as much as possible
- Keep data on the device as far as possible (using target data regions)
- GPUs need lots of threads to work efficiently
- Need to expose a lot of parallelism – much more than for the CPU
- Can program a GPU using OpenMP with a single pragma!

    !$omp target teams distribute parallel do

Some advanced topics that are not explored here in detail?

- Memory movement best practice
- Pointer swapping
- Halo exchange etc
- The Declare Target Directive
- Device Memory Functions
- The Device Pointer Clauses
- Profiling

List is quite long …

Technical differences between OpenMP/ACC

@courtsey: James Beyer and Jeff Larkin, Nvidia

Parallel: Similar, but different

OMP Parallel

- Creates a team of threads
- Very well-defined how the number of threads is chosen
- May synchronize within the team
- Data races are the the user’s responsibility

ACC Parallel

- Creates 1 or more gangs to workers
- Compiler free to choose number of gangs, workers, vector length
- May not synchronize between gangs
- Data races not allowed

Loop: Similar but different

OMP Loop (For/Do/

- Splits ("Workshares") the iterations of the next loop to threads in the team, guarantees the user has managed any data races
- Loop will be run over threads and scheduling of loop iterations may restrict the compiler

ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- User able to declare independence w/o declaring scheduling
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

Distribute vs Loop

OMP Loop (For/Do/

- Must live in a TEAMS region
- Distributes loop iterations over 1 or more thread teams
- Only master thread of each team runs iterations, until PARALLEL is encountered
- Loop iterations are implicitly independent, but some compiler optimizations still restricted

ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

Generate a 1 or more thread teams

Distribute “i” over teams.

No information about “j” or “k” loops

```
#pragma acc parallel


#pragma acc loop
      =0; i<n; i++)
#pragma acc loop
      (j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Distribute example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i
    for(j=0;j<m;j
      for(k=0;k<p;k++)
}
```

Generate a 1 or more gangs

These loops are independent, do the *right thing*

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Synchronization

OpenMP

- Users may use barriers, critical regions, and/or locks to protect data races
- It's possible to parallelize non-parallel code

OpenACC

- Users expected to refactor code to remove data races.
- Code should be made truly parallel and scalable

Synchronization example

```
#pragma omp parallel for

for (i=0; i<N; i++)

{

#pragma omp critical

  A[i] = rand();

  A[i] *= 2;

}
```

```
parallelRand(A);

#pragma acc parallel loop

for (i=0; i<N; i++)

{

  A[i] *= 2;

}
```

Closing thoughts

- Both **OPENMP** and **OPENACC** directive based to expose all available parallelism
- They are both similar but yet bit different in their approach
- Each approach has clear tradeoffs with no clear "winner"
- It should be possible to translate between the two, but the process may not be automatic
- Easily port your code on the GPU and could get good performance
- OpenACC is strongly supported by NVIDIA, which means the best performance could be achieved on the NVIDIA GPUS

@courtsey: Jeff Larkin, Nvidia

Tasks-4: C/C++ Code

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

- Iterate until converged
- Iterate across matrix element
- Calculate new value from neighbours
- Compute max error for convergence
- Swap input/output arrays

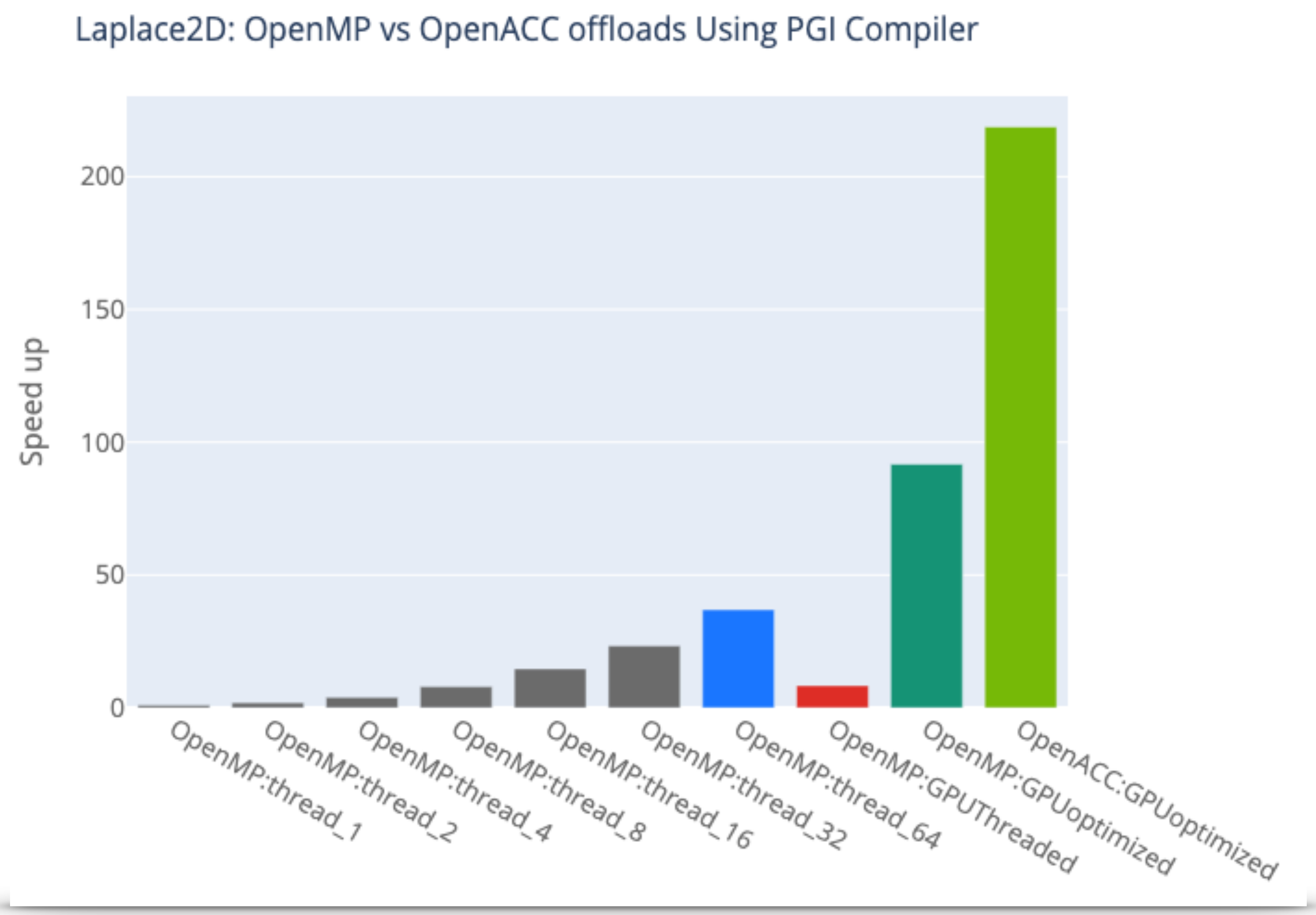
Accelerate serial code with OpenACC

- Add Kernels
- Parallel Loop
- Save execution time

Grazie Mille!!

Feel free to reach me out
n.shukla@cineca.it

Performance OpenMP Vs OpenACC


Laplace2D: OpenMP vs OpenACC offloads Using PGI Compiler
Speed up
OpenMP:thread_1
OpenMP:thread_2
OpenMP:thread_4
OpenMP:thread_8
OpenMP:thread_16
OpenMP:thread_32
OpenMP:thread_64
OpenMP:GPUThreaded
OpenMP:GPUoptimized
OpenACC:GPUoptimized