Porting OpenACC to OpenMP offloading

N. Shukla

CINECA HPC Summer School at Bologna Italy

Email: n.shukla@cineca.it

July 9-10th, 2024

House keeping

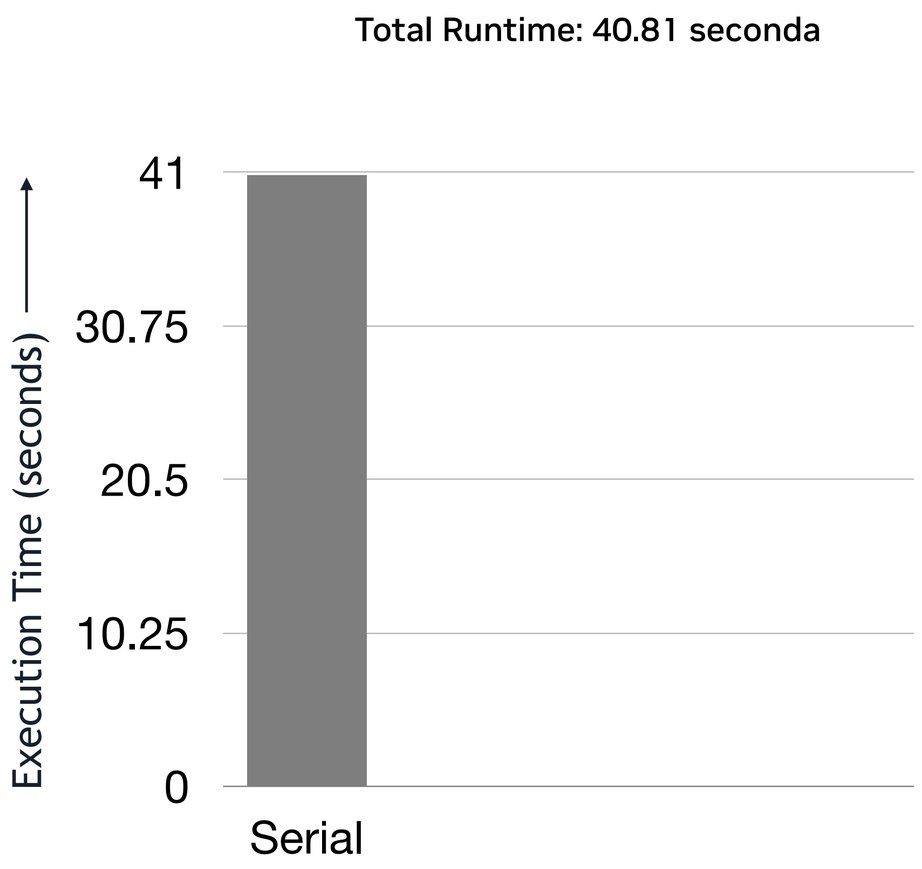
- Git clone
- cd Day-6th

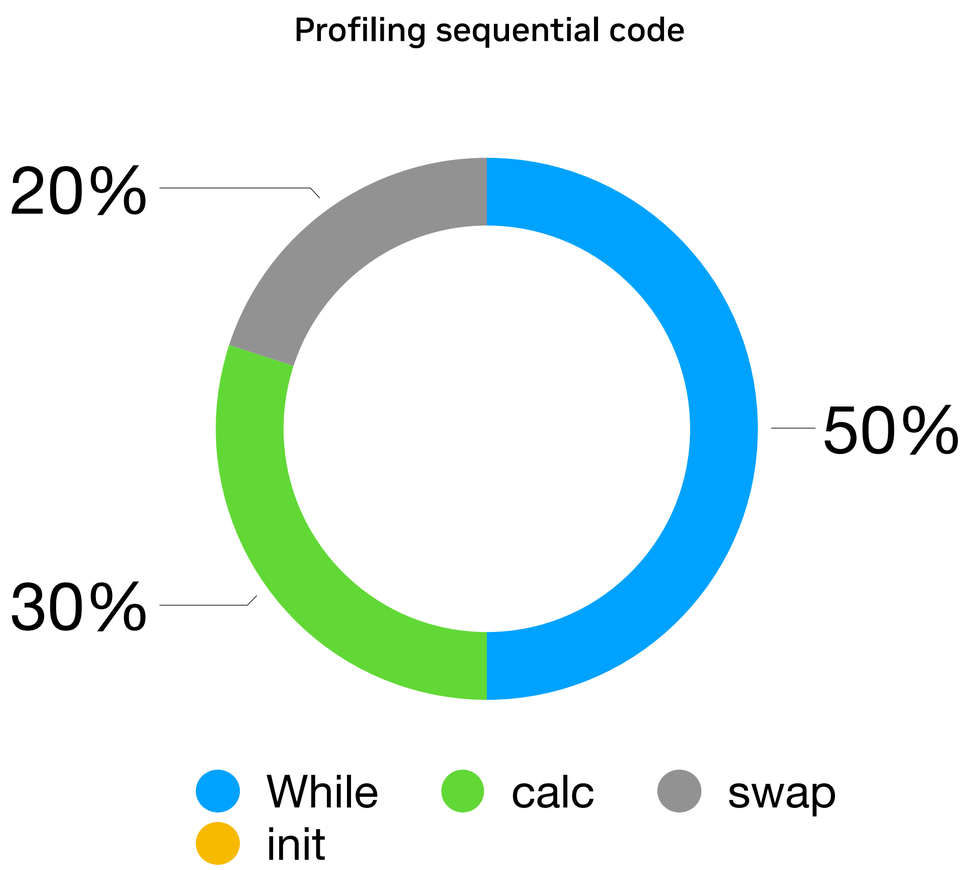
```
salloc -N1 --cpus-per-task=1 --ntasks-per-node=8 -A XX -t 00:00:00 -p
boost_usr_prod —qos=XX --gres=gpu:1
```

Asking Interactive node

Hotspots: Identify the portions of code that took the longest to run

Simulation was performed 1000 Iterations





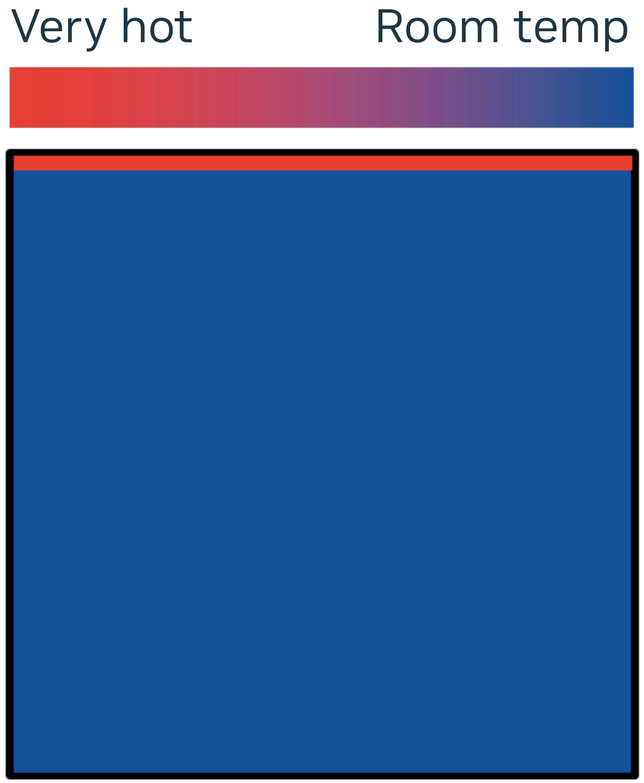
CINECA

Laplace heat transfer

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.

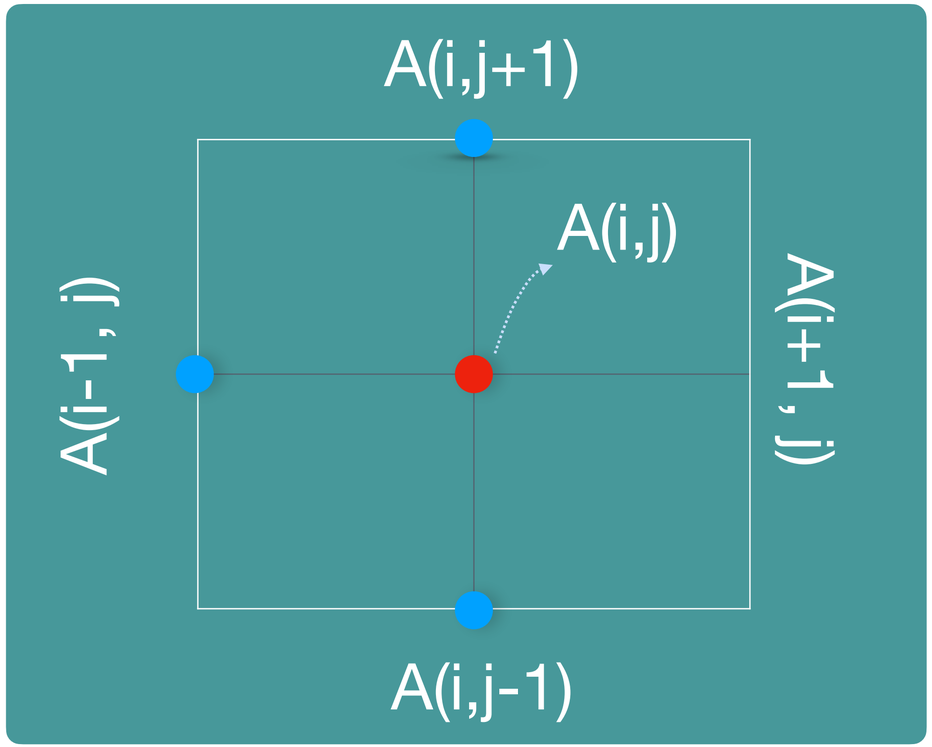


Code description

- Iteratively converges to correct value (e.g. Temperature)
- by computing new values at each point from the average of neighboring points.
- Example: Solve Laplace equation in 2D

$$\nabla^2 f(x,y) = 0$$

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$



Code description

```
while (error > tol && niter < niter_max)
error = 0.0;

 for (int j = 1; j < n-1; ++j) {
  for (int i = 1; i < m-1; ++i) {
   Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m]
   + A[idx+m]);

   error = fmax(error, fabs(Anew[idx] - A[idx])); }
   }
 for (int j = 1; j < n-1; ++j)
  for (int i = 1; i < m-1; ++i)
   A[j][i] = Anew[j][i]
```

Iterate until converged

Iterate cross matrix elements

Compute max error for converge

Compute max error for converge

Swap input/output arrays

laplace2d-openmp: parallelize with OpenMP

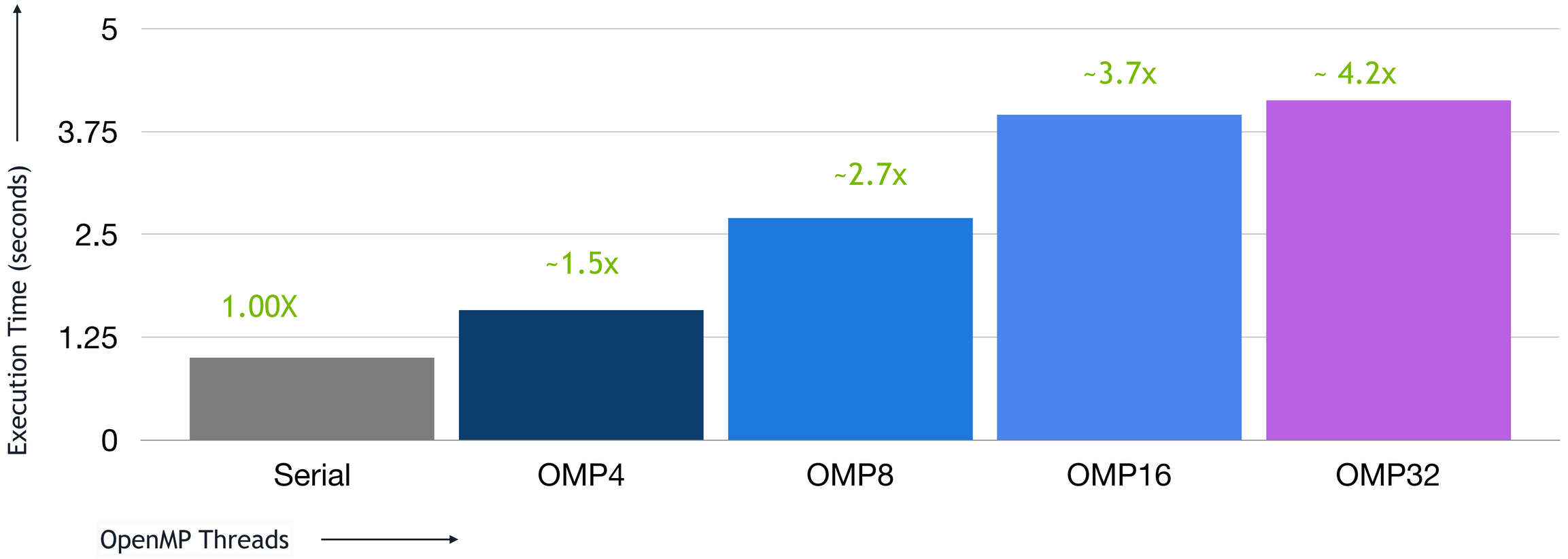
```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

laplace2d-openmp: parallelize with OpenMP

```
while ( error > tol && iter < iter_max ) {
  error=0.0;
#pragma omp parallel for collapse(2) shared(m, n, Anew, A) reduction(max:error)
  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]);
    }
  }
#pragma omp parallel for collapse(2) shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Performance speed up (higher is better)

Simulation was performed 1000 Iterations



AGENDA

- Why OpenMP for porting code?
- Thread creation on the GPU
- Data managent
- Using multiple GPUs
- Asynchronism
- Comparative study: OpenACC versus OpenMP

Why should choose OpenMP for porting scientific code on GPU?

Why porting code with OpenMP offloading?

Cross-platform Compatibility

- **Multi-Vendor Support:** supported by a wide range of compilers across different platforms and vendors, including GCC, Clang, Intel, and NVIDIA compilers
- **Versatility:** providing flexibility in deployment options

Widespread Adoption and Familiarity

- **Broad Usage:** widely adopted standard in the scientific and engineering communities
- **Existing Codebase:** extending these codes to GPUs using OpenMP can be more seamless

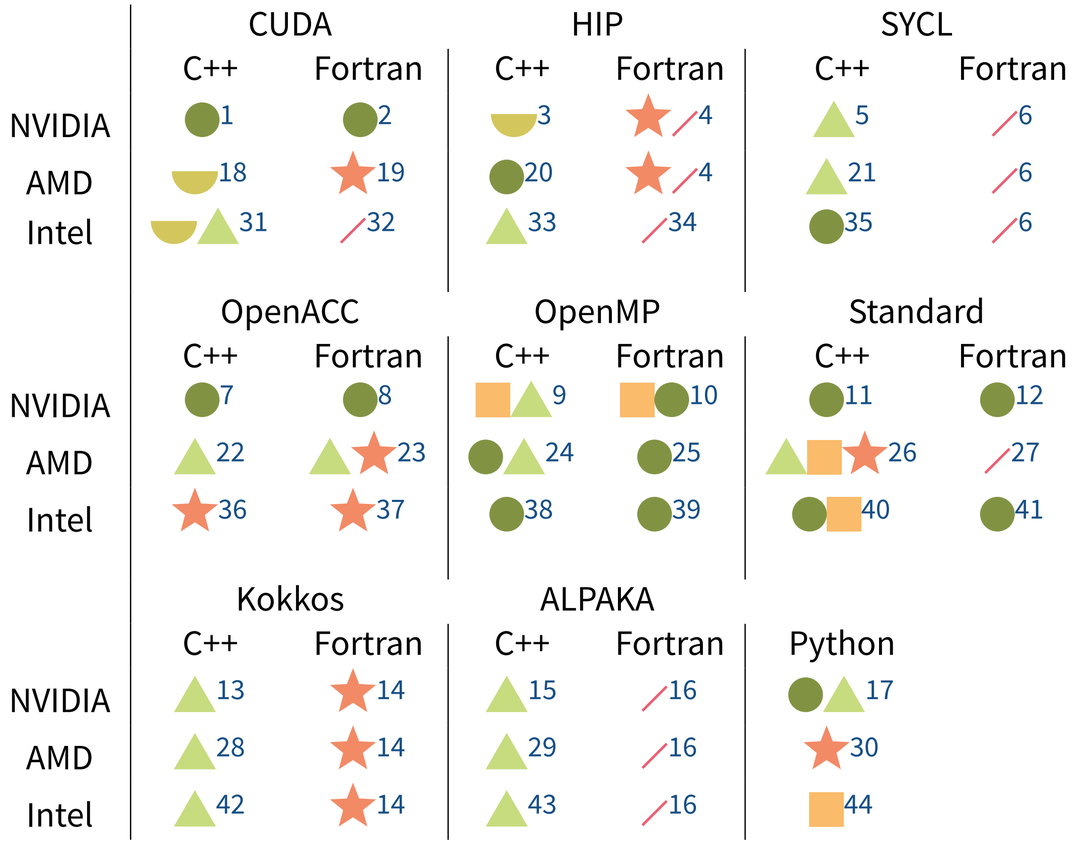
Unified Parallelism

- **Single Framework:** unified framework for parallelism across multiple platforms, including CPUs and GPUs
- **Interoperability:** simplifies the development process supporting parallelism on both CPU and GPU

Performance Optimization

- **Fine-Grained Control:** offers fine-grained control over parallel execution, including thread and memory management
- **Task Parallelism:** supports advanced features like task parallelism, which can be particularly beneficial for scientific applications with dynamic or irregular workloads

Why porting code with OpenMP offloading?



- State: Sep 2023 *evolving*
- Categories
  - Full vendor support
  - Indirect, but comprehensive support, by vendor
  - Vendor support, but not (yet) entirely comprehensive
  - Comprehensive support, but not by vendor
  - Limited, probably indirect support – but at least some
  - No direct support available
  - C++ C++ (sometimes also C)
  - Fortran Fortran

Thanks to @Andreas Hertena JSC

Compiler Support

OpenMP In Clang

Multi-vendor effort to implement OpenMP in Clang (including offloading)
Current status– interesting

How to get it- https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en

Thanks to @Jeff Larkin Nvidia

OpenMP In Clang

How to get it, our way

Step one – make sure you have: gcc, cmake, python and cuda installed and updated

Step two – Look at

http://llvm.org/docs/GettingStarted.html

https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en

Step three –

git clone https://github.com/clang-ykt/llvm_trunk.git

cd llvm_trunk/tools

git clone https://github.com/clang-ykt/clang_trunk.git clang

cd ../projects

git clone https://github.com/clang-ykt/openmp.git

Thanks to @Jeff Larkin Nvidia

OpenMP In Clang

How to build it

cd ..

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=DEBUG|RELEASE|MinSizeRel \
-DLLVM_TARGETS_TO_BUILD=“X86;NVPTX” \
-DCMAKE_INSTALL_PREFIX=“<where you want it>" \
-DLLVM_ENABLE_ASSERTIONS=ON \
-DLLVM_ENABLE_BACKTRACES=ON \
-DLLVM_ENABLE_WERROR=OFF \
-DBUILD_SHARED_LIBS=OFF \
-DLLVM_ENABLE_RTTI=ON \
-DCMAKE_C_COMPILER=“GCC you want used" \
-DCMAKE_CXX_COMPILER=“G++ you want used" \
-G "Unix Makefiles" \ !there are other options, I like this one
../llvm_trunk
make [-j#]
make install
```

Thanks to @Jeff Larkin Nvidia

OpenMP In Clang

How to use it

```
export LIBOMP_LIB=<llvm-install-lib>
export OMPTARGET_LIBS=$LIBOMP_LIB
export LIBRARY_PATH=$OMPTARGET_LIBS
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OMPTARGET_LIBS
export PATH=$PATH:<llvm_install-bin>
clang -O3 -fopenmp=libomp -omptargets=nvptx64sm_35-nvidia-linux …
```

Thanks to @Jeff Larkin Nvidia

Building and running enabled OpenMP code

Flags	Specification
-mp	enable OpenMP targeting device
-mp=gpu	to generate an executable that will run serially on the host CPU
–gpu=cc80	map OpenACC parallelism to an NVIDIA GPU, compile targeting compute capability
–gpu=managed	place all allocatables in CUDA Unified Memory
–gpu=pinned	use CUDA pinned memory for all allocatables
-Minfo=mp	Compiler diagnostics for OpenMP

```
$ nvc -mp=gpu -gpu=cc80,managed -Minfo=mp -o binary OpenMP_Code.c
```

Compiler support

How to use it

	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>`

Parallelize on the CPU

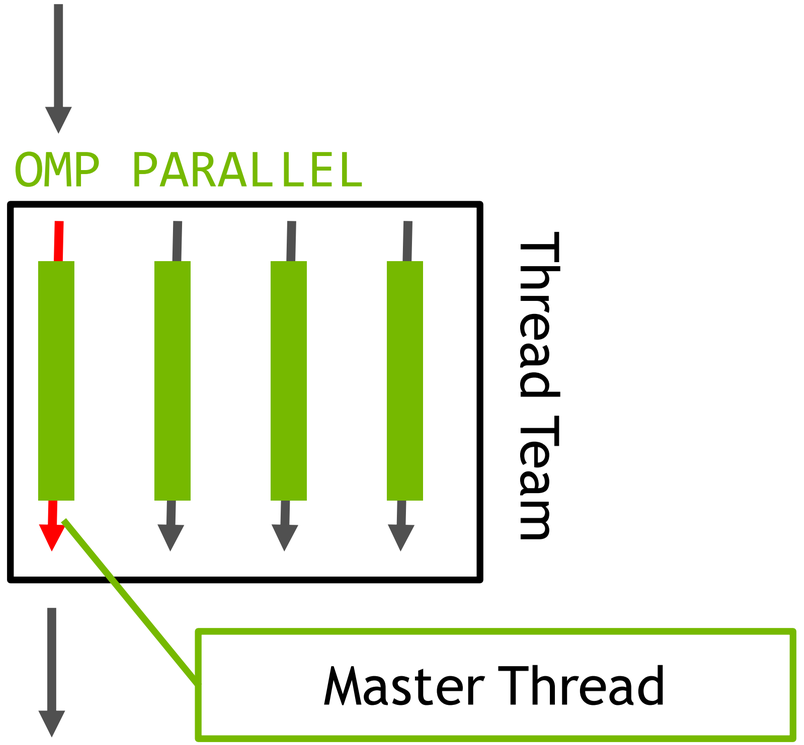
OpenMP Worksharing

PARALLEL Directive

Spawns a *team* of *threads*

Execution continues redundantly on all threads of the team.

All threads join at the end and the *master* thread continues execution.

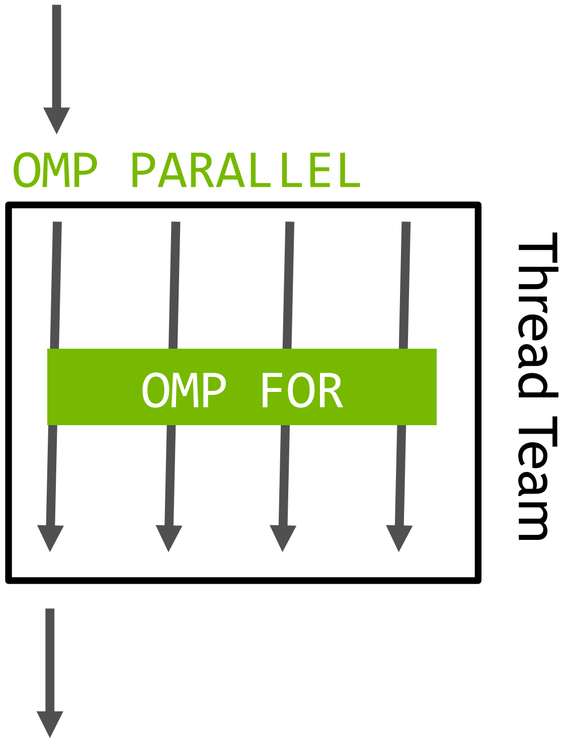


OpenMP Worksharing

FOR/DO (Loop) Directive

Divides (“*workshares*”) the iterations of the next loop across the threads in the team

How the iterations are divided is determined by a *schedule*.



CPU-Parallelism

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

#pragma omp parallel for reduction(max:error)
    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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

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

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

← Create a team of threads and workshare this loop across those threads.

← Create a team of threads and workshare this loop across those threads.

CPU-Parallelism

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

#pragma omp parallel
{
#pragma omp for reduction(max:error)
    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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp barrier
#pragma omp for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

← Create a team of threads

← Workshare this loop

← Prevent threads from executing the second loop nest until the first completes

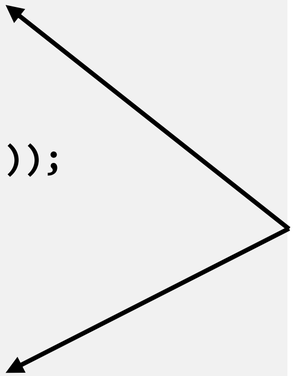
CPU-Parallelism

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

#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

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

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```



Some compilers want a SIMD directive to *simdize* on CPUS.

Targeting to GPU

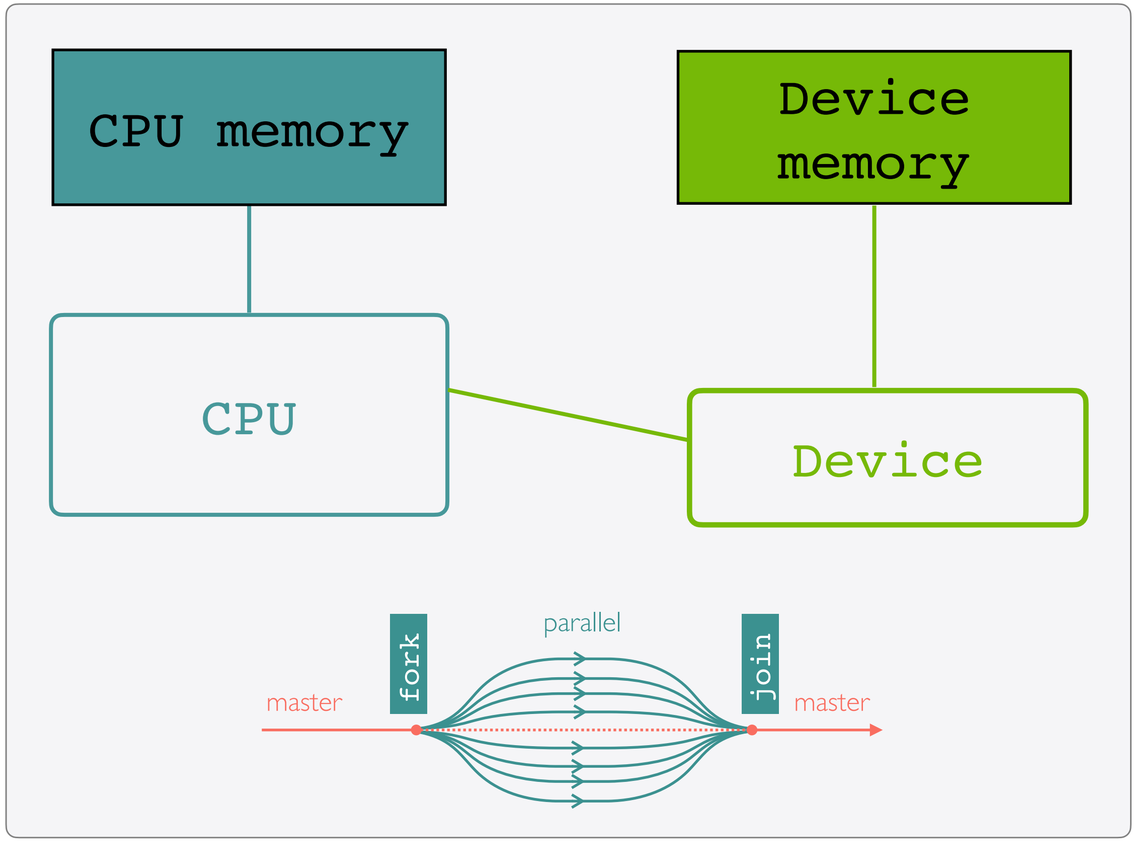
OpenMP uses host-device model

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?

TARGET Directive: Offloads execution and associated data from the CPU to the GPU

- 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 […] …]
```

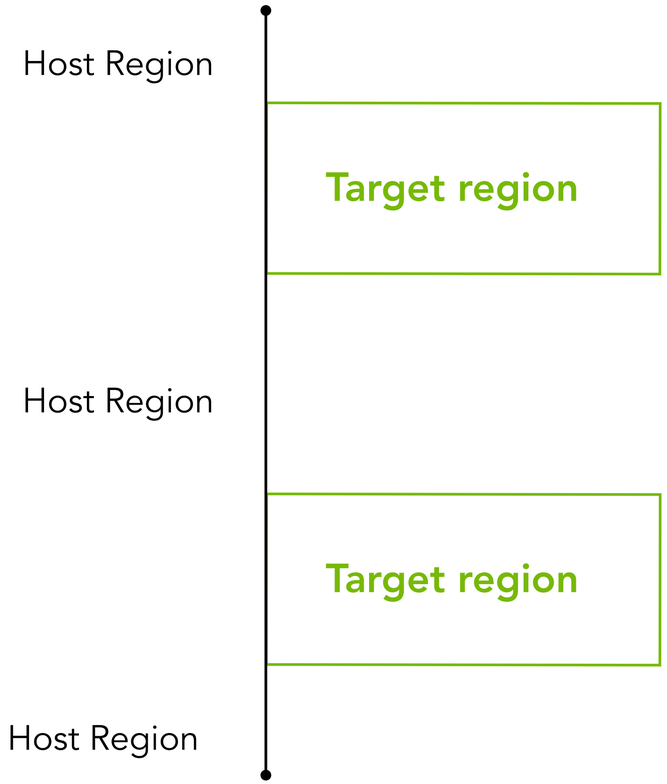
Target clause gets kennel code running on the device

```
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)

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

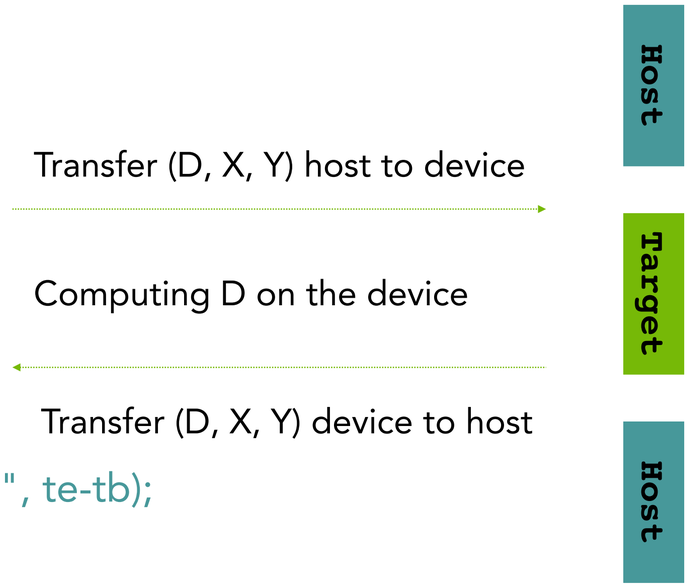
OpenACC is the default way that data is handled when entering a parallel work region.

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);
}
```



Managing data movement

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

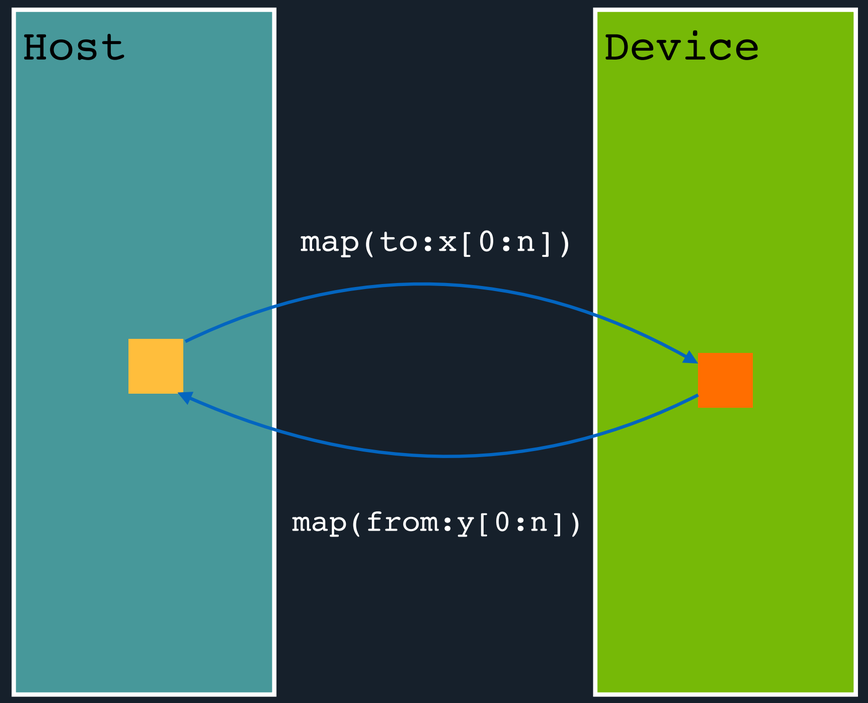
"The map clause specifies how an original list item is mapped from the current task's data environment to a corresponding list item in the device data environment of the device identified by the construct."

- 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

/*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

```
1   !$omp target enter data map(to: A, B, C)
2   !$omp target
3   ... ! Use A, B and C on device
4   !$omp end target
5
6   ! Copy A from device to host
7   !$omp target update from(A(1:N))
8
9   ! Change A on the host
10  A = 1.0
11
12  ! Copy A from host to device
13  !$omp target update to(A(1:N))
14
15  !$omp target
16  ... ! Use A, B and C on device
17  !$omp end target
18
19  !$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 à 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 à host

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

#pragma omp target update to(A)          //Updates A Host à 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
```

CINECA

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

Device clause

- Specify which device to offload to in a multi-device environment
  #pragma omp target device(i)
- Device number an integer
  Assignment is implementation-specific
  Usually start at 0 and sequentially increments
- Works with target, target data, target enter/exit data, target update directives

Expressing parallelism

Heterogeneous Program Execution

The target construct is a task generating construct which transfers the control flow to the target device

- Transfer of control is sequential and synchronous

OpenMP separates offload and parallelism

- Explicitly parallel regions are created on the target
- Combined with any OpenMP construct

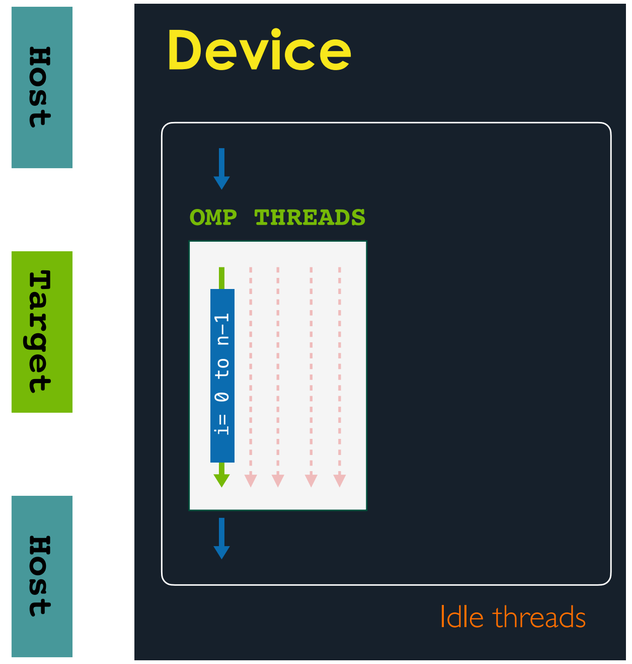
OpenMP GPU Offload support all “normal” OpenMP constructs

- E.g. parallel, for/do, barrier, sections , tasks etc
- Not every construct will be useful

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);
}
```



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

Support multi-level parallel device

Syntax (C/C++):

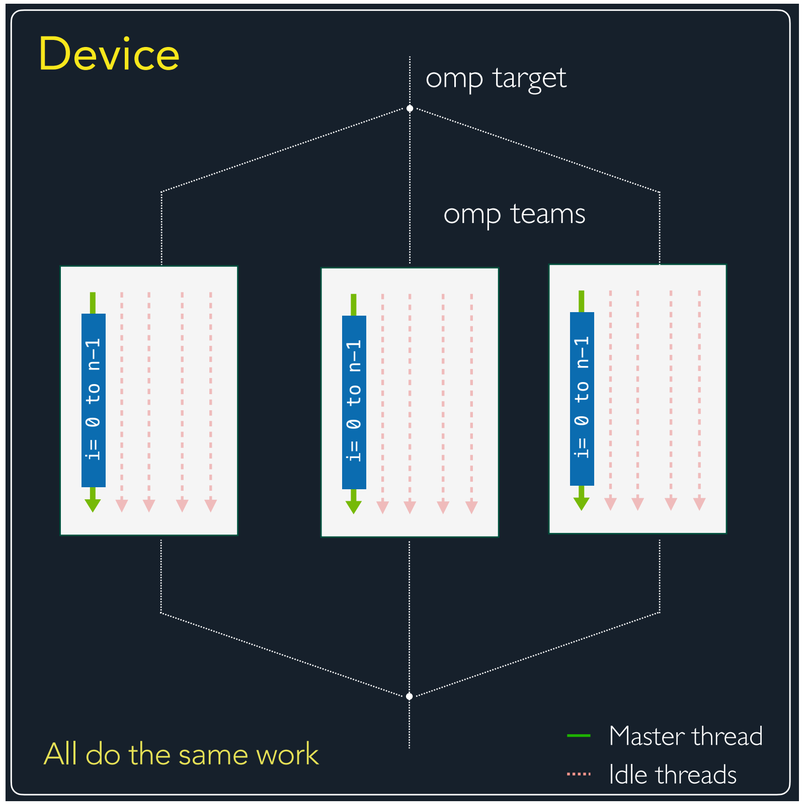
```
#pragma omp teams [clause[[,] clause],...]
```

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)



Distribute constructs shares works across the teams

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

Support multi-level parallel device

Syntax (C/C++):

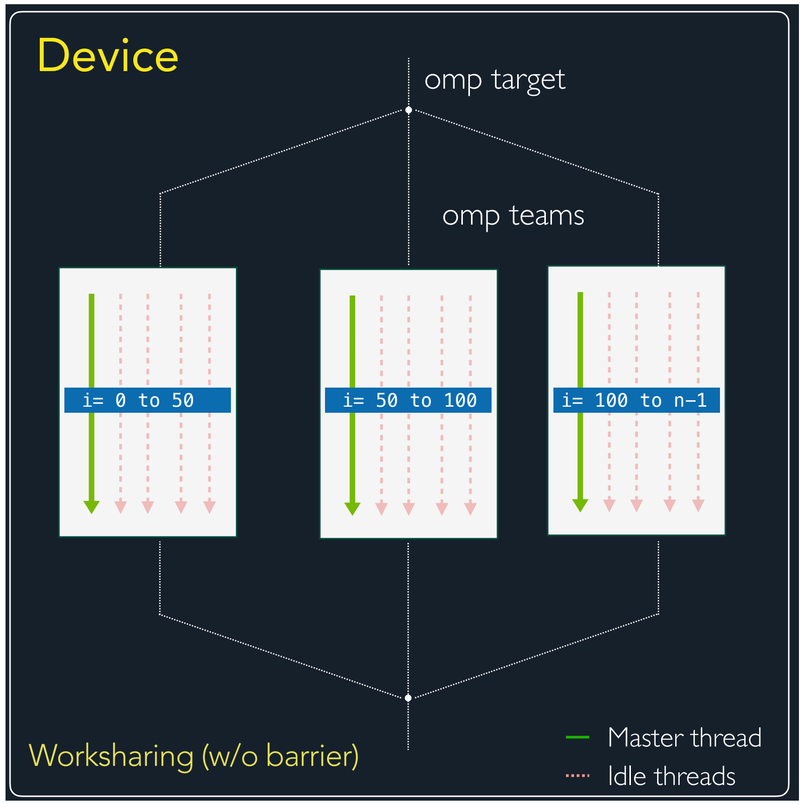
```
#pragma omp teams [clause[[,] clause],...]
```

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)



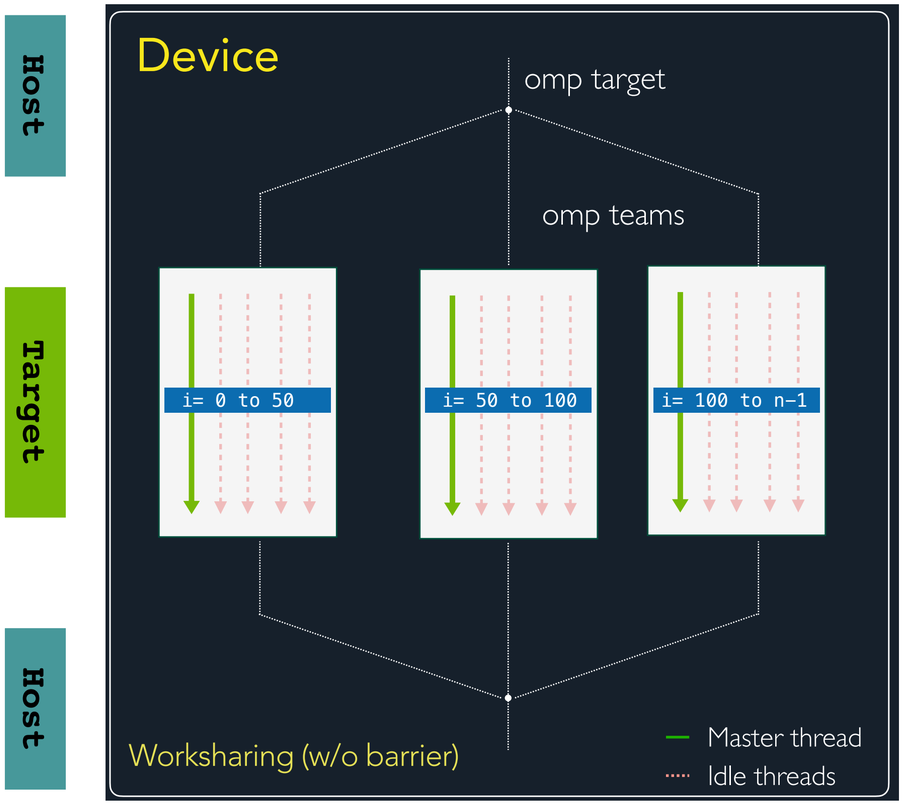
Distribute constructs shares works across the teams

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[
  }

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



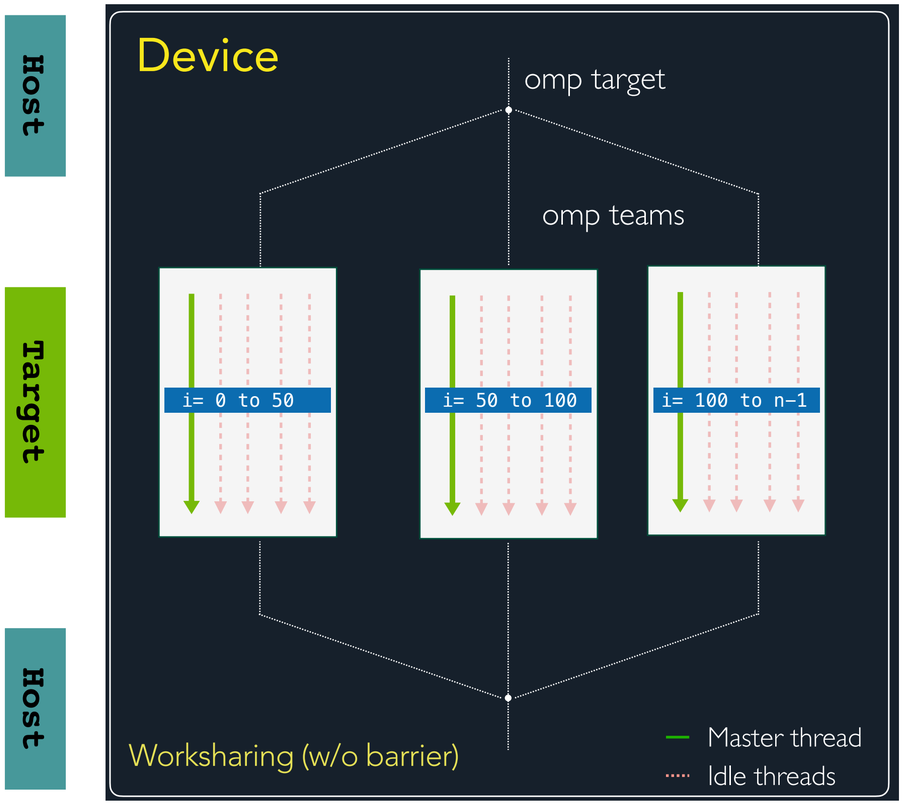
Parallel for shares work to all threads of the teams

Launches threads within the team and the do distributes iteration across the threads in a team

```
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[
  }

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

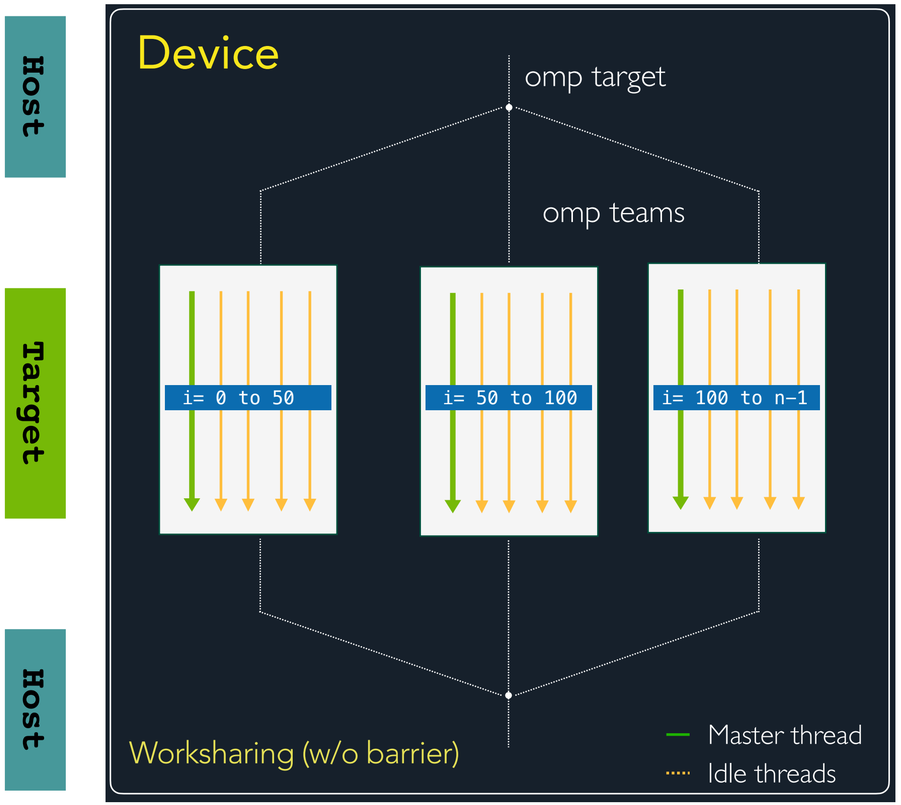


Expressing Parallelism: SIMD

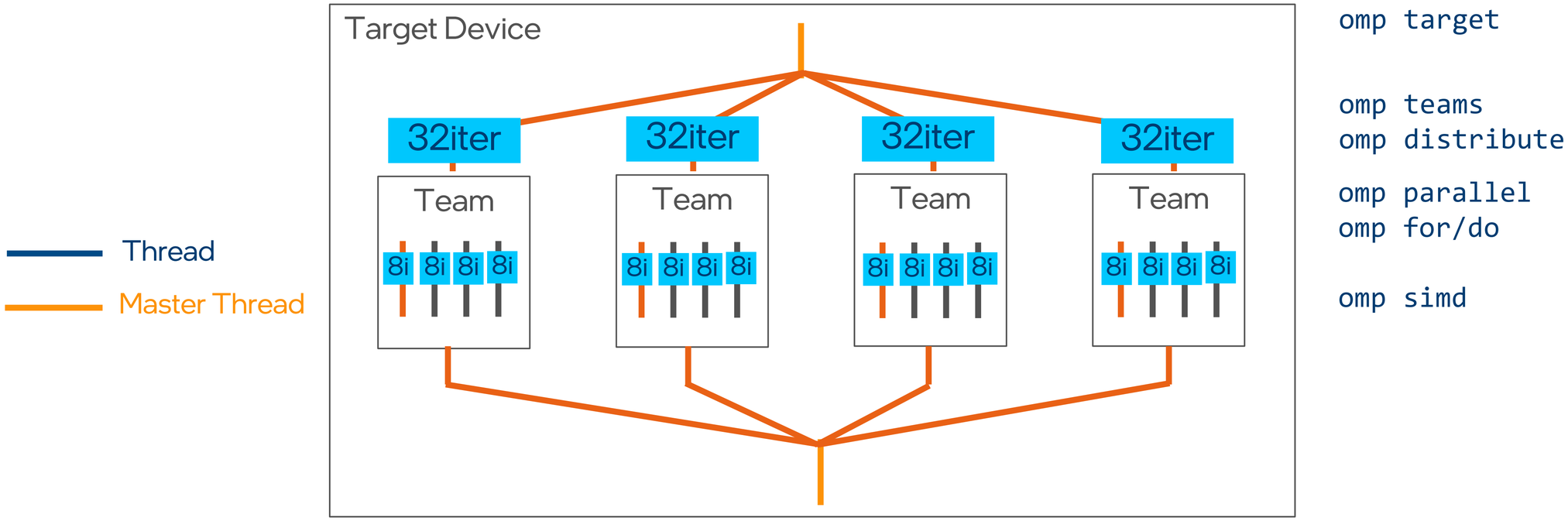
```
void daxpygpu( float A, double *D, double *X, double *Y, size_t n)
{

  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 simd
  for (i = 0; i < n; i++)
         D[i] = A*X[i] + Y[
  }

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



Multi-level parallel parallelism




```
omp target

omp teams
omp distribute

omp parallel
omp for/do

omp simd
```

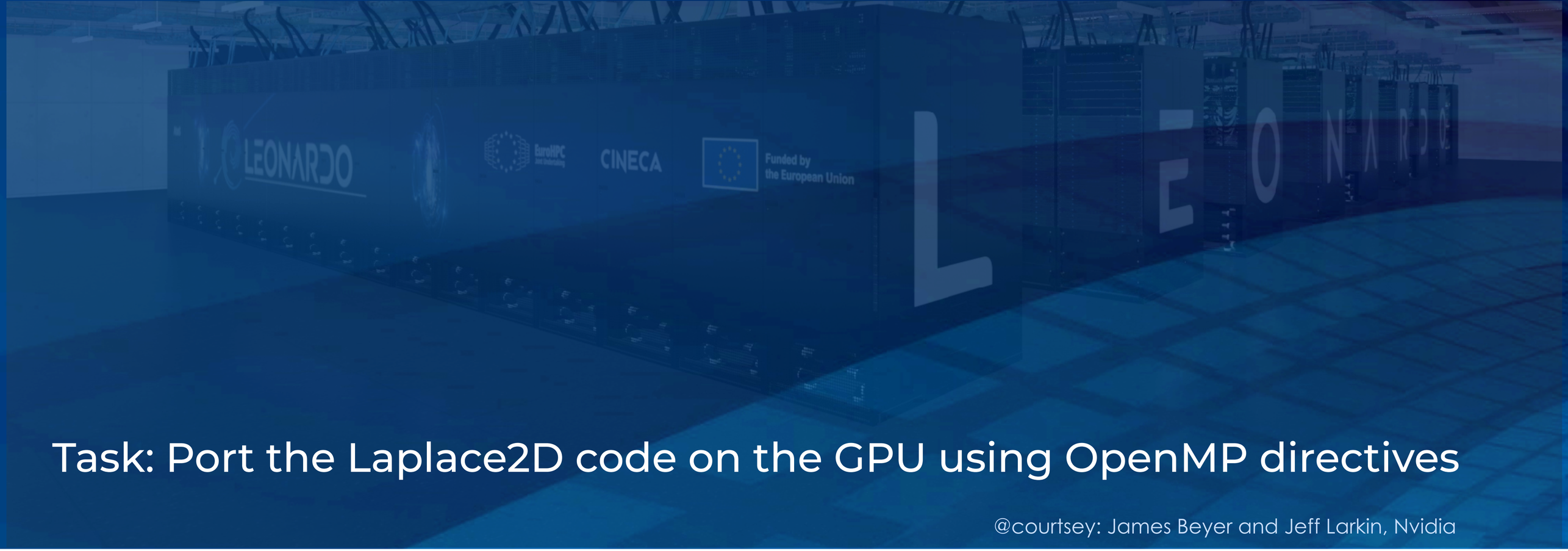
Combined constructs

- **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



Task: Port the Laplace2D code on the GPU using OpenMP directives

@courtsey: James Beyer and Jeff Larkin, Nvidia

Step-1: Target the GPU

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target map(alloc:Anew[:n+2][:m+2]) map(tofrom:A[:n+2][:m+2])
{
#pragma omp parallel for reduction(max:error)
    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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Moves this region of code to the GPU and explicitly maps data.

Step-2: Target the GPU

```
#pragma omp target data map(alloc:Anew) map(A)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp target teams distribute parallel for reduction(max:error)
        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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
```

Explicitly maps arrays for the entire while loop.

- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.

Step-3: Splitting Teams & Parallel

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

#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Distribute the “j” loop over teams.

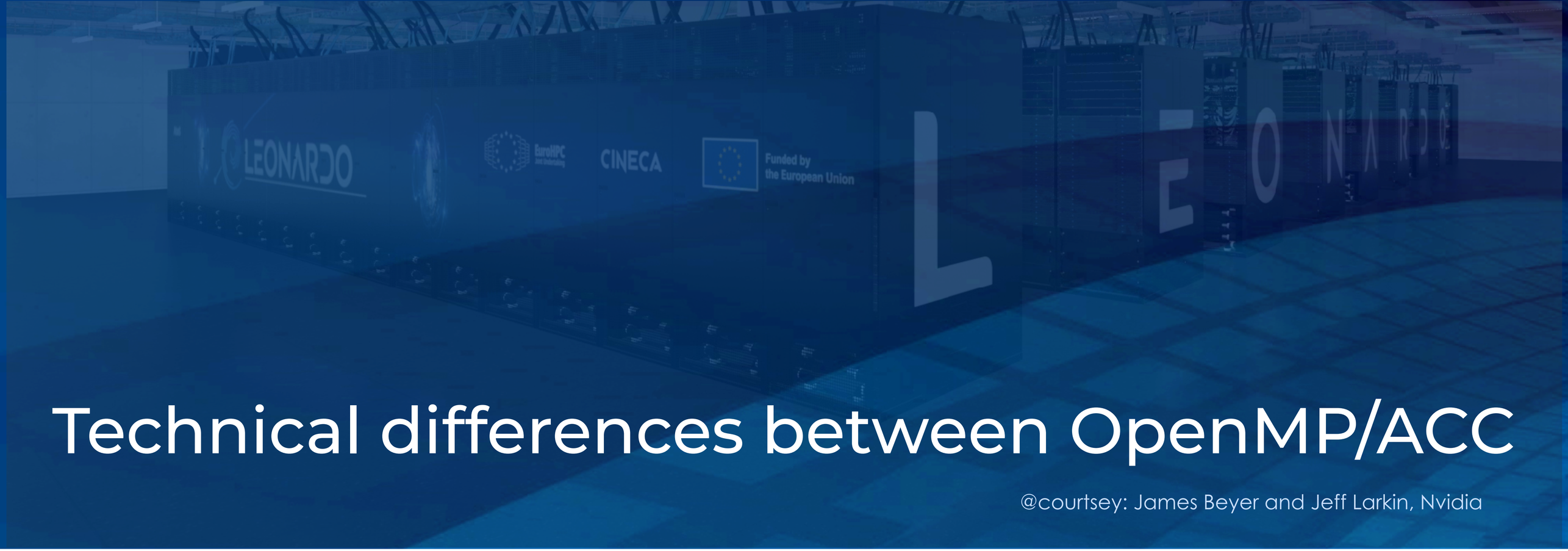
Workshare the “i” loop over threads.

Step-4: Improved Schedule (Collapse)

```
#pragma omp target teams distribute parallel for \
  reduction(max:error) collapse(2) schedule(static,1)
        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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for \
  collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Assign adjacent threads adjacent loop iterations.



Technical differences between OpenMP/ACC

@courtsey: James Beyer and Jeff Larkin, Nvidia

OpenX (X = OMP, ACC): 1997-2021

Much a like, directive approach to accelerator for Fortran and C/C++ codes

OpenMP: prescriptive nature

Programmer explicitly parallelizes the code

- Requires to perform requested parallelization and little/no analysis by the compiler

Reproducibility

- Parallelization will be performed the same way whatever the hardware the code runs on

Substantially different architectures require different directives

- To perform optimally on multiple different architecture, one has to write different sets of directives for different architectures
- Sometimes, it requires change in the code, for example it may be beneficial to switch the order of loops
- Fairly consistent behaviour between implementations

OpenACC: descriptive nature

Rely on compiler

- Compiler parallelises the code with guidance from the programmer

Compiler takes decision

- the code may parallelize or even may not be parallelized

Compiler executes information from the programmer and heuristics about the architecture to make decision

- The same code, compiled to run on GPU, or on Xeon Phi, or on CPU, may therefore yield different binary code
- Different compilers may yield difference performance
- Quality of implementation greatly affects the results

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

Compiler-driven mode

OMP Parallel

- Fully user-driven (no analogue)
- Some compilers choose to go above and beyond after applying OpenMP, but not guaranteed

ACC Parallel

- Kernels directive declares desire to parallelize a region of code, but places the burden of analysis on the compiler
- Compiler required to be able to do analysis and make decisions

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 vs Loop

```
#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++)
}
```

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++)
}
```

What's the *right thing?*

Interchange? Distribute? Workshare?
Vectorize? Stripmine? Ignore? …

```
#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