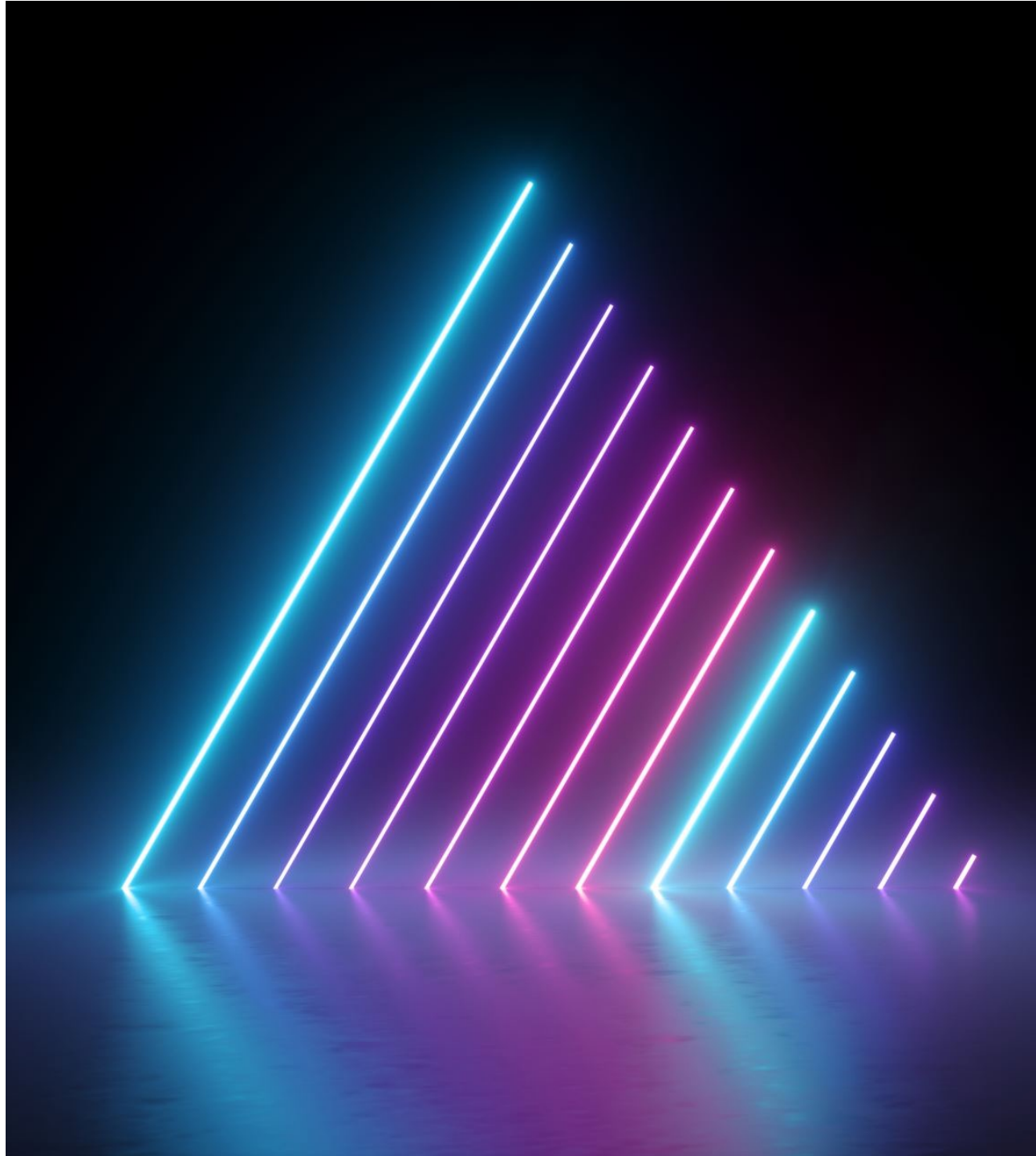
OpenMP Offloading Basics

by

Kent Milfeld (TACC)

Emanuele Vitali (CSC)



https://www.nersc.gov/assets/Uploads/OLCF-OMP-Day2.pdf

Preface

- Thanks to Henry Jin (NASA Ames) and Swaroop (ORNL) and (CSC/EuroHPC JU) for slides and content collaboration.

- Slides and Presentation assume:
  - Some experience or basic knowledge of OpenMP
  - Working knowledge of C | C++ | F90

- Spec: openmp.org/specifications (OpenMP API 5.2 Specification)
- Examples: openmp.org/specifications (OpenMP API 5.2.1 Examples)
  github.com/openmp/examples
  (codes in sources directory of each chapters)

https://github.com/csc-training/summerschool/tree/master/gpu-openmp

Session Objective

Learn about:

- What OpenMP Offloading is
- Basic Operations of Offloading
- Other Offload approaches (Cuda/OpenACC)
- OpenMP Offload Execution Model
- Compiling options for OpenMP Offload
- OpenMP Conditional code eliding/execution
- Some APIs
- Cuda/OpenMP comparison of simple code
- Target Directive (Offload)
- Teams Directive
- Worksharing Directives

What is OpenMP Offloading

- Set of OpenMP constructs for heterogeneous systems
  - **GPUs**, FPGAs, …
- Code regions are offloaded from the host CPU to devices.
  - High-level abstraction layer for GPU programming
- In principle same code can be run on various systems
  - CPUs only
  - NVIDIA GPUs, AMD GPUs, Intel GPUs, ...
- Standard defined for C/C++ and Fortran

What are the basic OpenMP offload operations?

- Transfer of compute-intensive tasks to separate processor, (device),hardware accelerator (usually GPUs), grid, cloud, etc.

- Implies the following for the device for discrete (separate) memory on host and device:
  - Creating executable (task)
  - Creating/Freeing memory on the device
  - Marshalling and Transferring data to/from device
  - Launching task

There is much more!

OpenMP vs. OpenACC

- OpenACC ihas similar compiler directive-based approach for GPU programming
  - It is an open standard; however, NVIDIA is the major driver.

- Why OpenMP and not OpenACC?
  - OpenMP broader platform and compiler support
  - OpenACC support for AMD GPUs is limited
  - OpenACC can provide better performance in NVIDIA GPUs

OpenACC support for AMD GPUs

- OpenACC support for AMD GPUs in GNU compiler under development
- OpenACC support in general for Clang/Flang is under development
- Cray compilers
  - Fortran compiler supports OpenACC v2.7, support for latest OpenACC coming
  - C/C++ compiler does not support OpenACC
- In LUMI, only Fortran is supported with OpenACC
- For now, OpenACC is not a recommended approach for new codes targeting AMD GPUs
  - if a Fortran code already uses OpenACC, it may be possible to use it

OpenMP vs CUDA/HIP

- Why OpenMP and not CUDA/HIP?
  - easier to start shifting work to GPUs (less coding)
  - simple things are simpler
  - same code can be compiled to CPU and GPU versions easily

- Why CUDA/HIP and not OpenMP?
  - can access all features of the GPU hardware
  - better control and assurance it will work as intended
  - more optimization possibilities

AMD and NVIDIA Terminology -- FYI

AMD	NVIDIA
Work-items or Threads	Threads
Workgroup	Block
Wavefront	Warp
Grid	Grid

Runs as a SIMD unit of 64 and 32 threads, respectively.

https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#amd-gpus

OpenMP execution model

- Host-directed execution with an attached accelerator
  - large part of the program is usually executed by the host
  - computationally intensive parts are **offloaded** to the accelerator

- Accelerator can have a separate memory
  - OpenMP exposes the separate memories through **data environment** that defines the memory management and needed copy operations

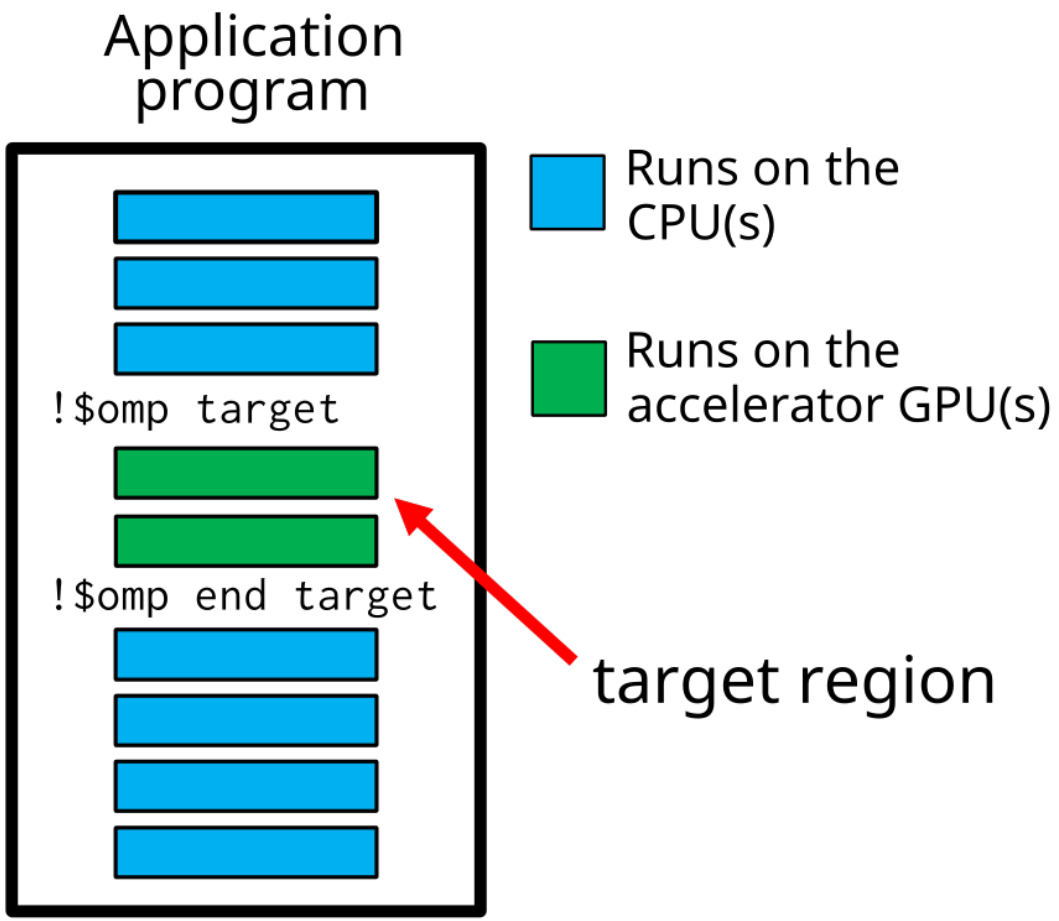
OpenMP execution model

- Host-directed execution with an attached accelerator
  - large part of the program is usually executed by the host
  - computationally intensive parts are **offloaded** to the accelerator

- Accelerator can have a separate memory
  - OpenMP exposes the separate memories through **data environment** that defines the memory management and needed copy operations

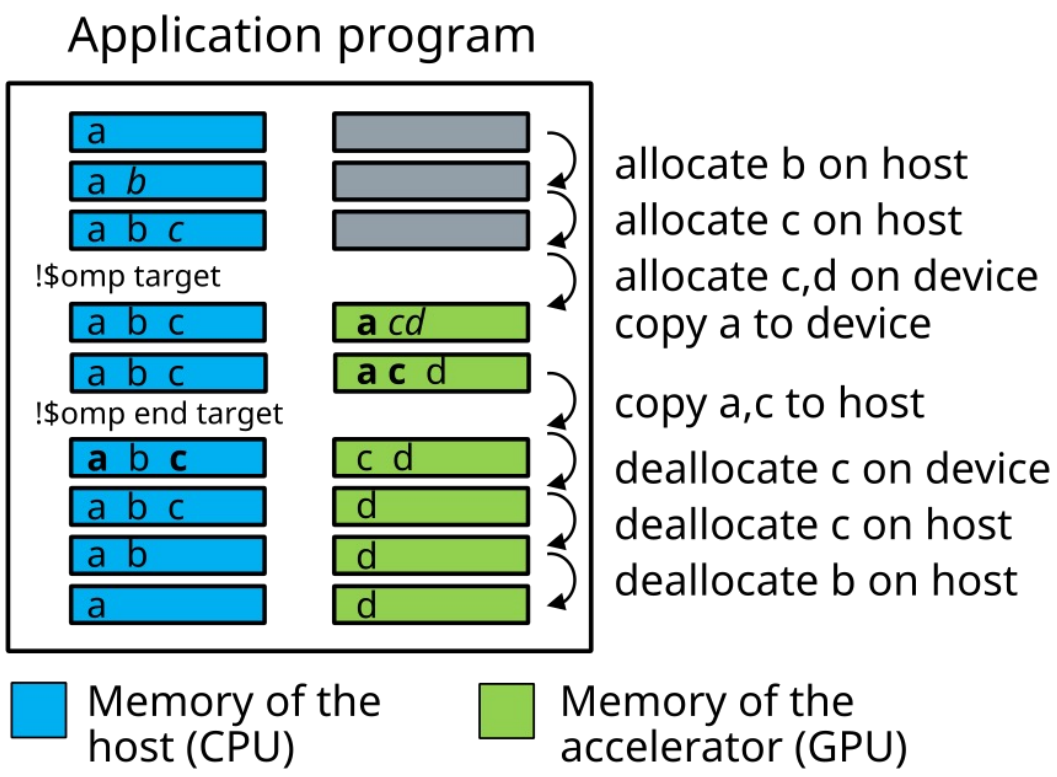
OpenMP execution model

- Program runs on the host CPU
- Host offloads compute-intensive regions (*kernels*) and related data to the GPU
- Compute kernels are executed by the GPU



OpenMP data model in offloading

- If host memory is separate from device memory
  - host manages memory of the device
  - host copies data to/from the device
- When memories are not separate, no copies are needed (difference is transparent to the user)



Compiling: target architecture options -- LUMI

LUMI specific: just load Cray Prog. Env. accelerator (and CPU) target modules

```
ml craype-accel-amd-gfx90a
ml craype-x86-trento
CC -fopenmp  gpu_code.cpp
```

GPU target

CPU target

NO target options requires on compile line

Must always specify the OpenMP option

LUMI specific: must first set up LUMI modules for computing on G partition

```
ml LUMI/23.03
ml partition/G
```

Compiling: general target options are required

Offloading may require architecture options
LLVM/NVIDA/GNU/ compilers may require a target triplet

```
<Architecture>-<Vendor>-<OS>*
```

and/or arch specification

```
<arch>
```

* See, for example: https://llvm.org/docs/AMDGPUUsage.html#amdgpu-target-triples
Cray: -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a

Compiling: general target options are required

- Compilers are moving toward auto-detection; but cross compiling will probably require options
  - Without the device options only host (CPU) versions are created and run!

Compiler	Device Options for Offload
NVIDIA	-mp=gpu (-gpu=cc##)
Cray	-fopenmp-targets=xx –Xopenmp-target=xx
Clang	-fopenmp-targets=xx
GCC	-foffload=yy

[e.g. cc##=cc70; xx=nvptx64 (or triplet such as nvptx64-nvida-cuda triplet; yy=amdgcn-amdhsa="-march=gfx90”,...]

Conditional compiling/execution for OpenMP

- Conditional compiling with _OPENMP macro:

```
#ifdef _OPENMP
   device specific code
#else
   host code
#endif
```

- **if** clause on some directives:
  can direct runtime to **ignore or alter behavior of construct**.

- metadirective and declare variant directives
  provide **conditional selection of directives and routines**

OpenMP internal control variables

- OpenMP has internal control variables
  - `OMP_DEFAULT_DEVICE` controls which accelerator is used.
- During runtime, values can be modified or queried with `omp_<set|get>_default_device`
- Values are always re-read before a kernel is launched and can be different for different kernels

Useful API routines

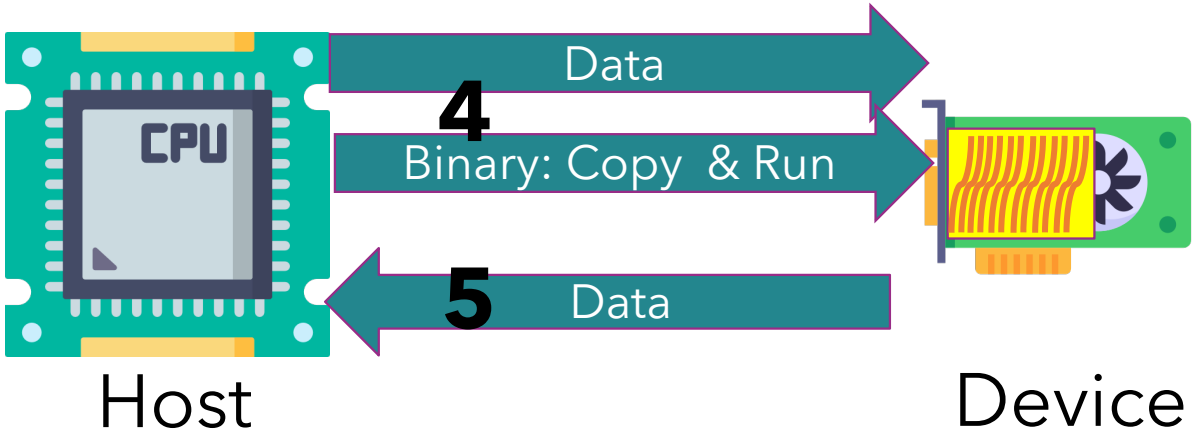
- `omp_is_initial_device()`
  - returns True when called in host, False otherwise
- `omp_get_num_devices()`
  - returns the number of devices available
- `omp_get_device_num()`
  - returns the number of the device where the function is called
- `omp_get_default_device()`
  - returns the default device
- `omp_set_default_device(n)`
  - sets the device n as default device

offload: run this code on another device (GPU)

```
main(){                                  1
  host_dat
  allocdev(dev_dat)
  cp2dev(host_dat, dev_dat)
  func_dev(...)
  cpback(host_dat, dev_dat)
}
```

2 compile prog.c

3 ./a.out



1.) Specify in Host program: code to be executed on Device (GPU) (and location in program where it is to be launched), how copies of Host data are to be made on Device, and returned.

2.) Compile: make fat binary with Host (CPU) and Device (GPU) binaries.

3.) Execute binary on Host

4.) Copy data and binary to device and run

5.) Copy data back from device.

Pseudo Code operations for Offloading a function

```
Declare Device function
                                        Create & Init host data
Create  device storage
Copy to device storage from host data

Launch device function          (& wait for completion)

Copy   data (results) back to host storage

Free device space
```

cuda function “offload”

Code Operations

```
          void vector_add(int n, float *x){ ... }; //declare dev function
int main(){
    int N<<28; float *d_x, h_x[N];                  // dev storage ptr(d_x)

    for(int i=0;i<N;i++)h_x[i]=1.0f; //host init

                                                    //create dev storage
                                                    //cp to dev

    vector_add(N,d_x);                          //Kernel launch w. params
                                                //Wait 4 Kernel 2 complete
                                                //cp dev d_x to host h_x


                                                //free dev mem
}
```

cuda function “offload”

Don’t be concerned with details of API routines.

```
__global__ void vector_add(int n, float *x){ ... }; //declare dev function
int main(){
    int N<<28; float *d_x, h_x[N];                  // dev storage ptr(d_x)

    for(int i=0;i<N;i++)h_x[i]=1.0f; //host init

    cudaMalloc((void**)&d_x, 4*N);                  //create dev storage
    cudaMemcpy(d_x,h_x,4*N,cudaMemcpyHostToDevice); //cp to dev

    vector_add<<<256,(N+256)/256>>>(N,d_x);  //Kernel launch w. params
    cudaDeviceSynchronize();                 //Wait 4 Kernel 2 complete
                                             //cp dev d_x to host h_x
    cudaMemcpy(h_x, d_x, sizeof(float)*N, cudaMemcpyDeviceToHost);

    cudaFree(d_x);                           //free dev mem
}
```

OpenMP directive based offload

Directive detail later.

```
void vadd(int n, float *x){ ... };
#pramga omp declare target (vadd) //declare dev function
int main(){
   int N<<28; float x[N];

   for(int i=0;i<N;i++) x[i]=1.0f; //host init
                                   //Create dev x & CP to
   #pragma omp target teams map(tofrom: x) //Launch wo params
   vadd(N,x);
                                   //CP from & free dev x
} // Simple cases often hide complexities!
```

more on Concepts

- OpenMP enables directive-based programming of accelerators with C/C++ and Fortran
- Host–device model
  host offloads computations to the device
- Host and device may have separate memories
  host controls copying into/from the device
- Key concepts:
  target -- launch an execution region on a device
  teams -- executes a region with multiple teams (on device CU/SMs)

target Construct

- OpenMP target construct specifies a region to be executed on GPU
- -- initially, runs with a single thread
- By default, execution in the host continues only after target region is finished
- May trigger implicit data movements between the host and the device

C/C++

```
#pragma omp target
{
  printf("1 thread on device\n");
}

#pragma omp target
  printf("hello again\n");
```

directive presented as pragma

F90

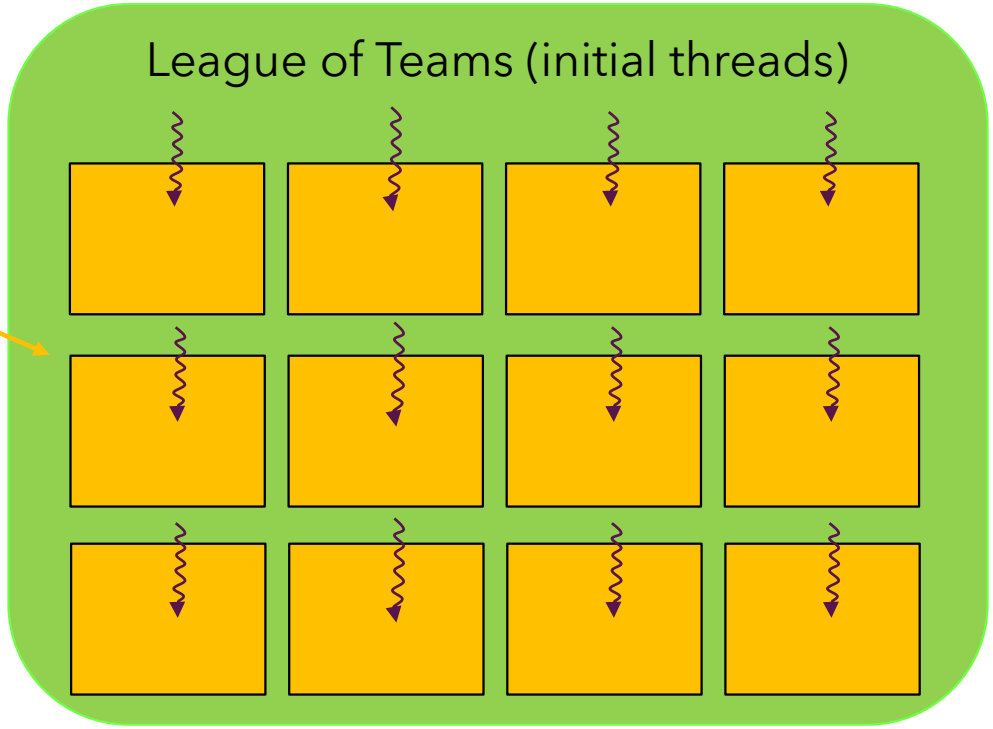
```
!$omp target

  print*, "1 thread on device"
!$omp end target
```

directive presented as comment

teams Construct

A CU (compute engine, amd) or
SM (streaming multiprocessor, nvidia)
is an independent execution unit,
supporting a set of threads.

An OpenMP team is an independent unit of execution of the region (for a CU/SM) which has an initial thread (task).




```
#pragma omp target teams
  printf("1 thread on device\n");
```

teams Construct

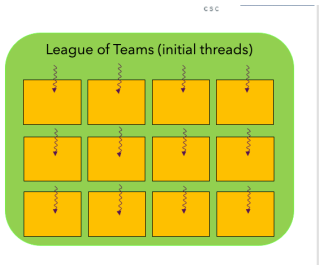
- OpenMP teams construct forms 1 or more teams, each with an initial thread*
  (the initial thread and its descendent threads are a contention group).
- teams directive is often combined with target as a single combined directive

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

```
!$omp teams
  !!code block
!$ omp end target
```

```
#pragma omp target
#pragma omp teams
{  // code executes on device  }
```

```
!$omp target
!$omp teams
  !! code execute on device
!$ omp end teams
!$ omp end target
```




```
#pragma omp target teams
{  // code executes on device  }
```

```
!$omp target teams
  !! code execute on device
!$ omp end target teams
```

Combined Construct

* The teams directive is general, it can also be used on a CPU to create a team of threads for each socket.

target teams Construct

run the next block, statement or function on the GPU as a single thread for each of 4 (max) teams.

C/C++

```
#pragma omp target teams num_teams(4)
{
    printf("%d\n",omp_get_team_num());
}
```

F90

```
!$omp target teams num_teams(4)

    print*, omp_get_team_num()

!$omp end target teams
```

*Without num_teams(), number of teams is implementation defined.

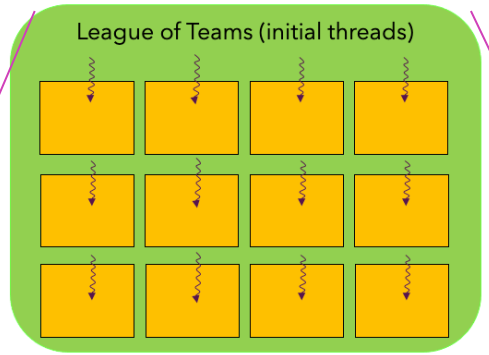
target teams parallel Construct

- A team of threads is created when teams is combined with the parallel directive.
- Number of threads is implementation defined.

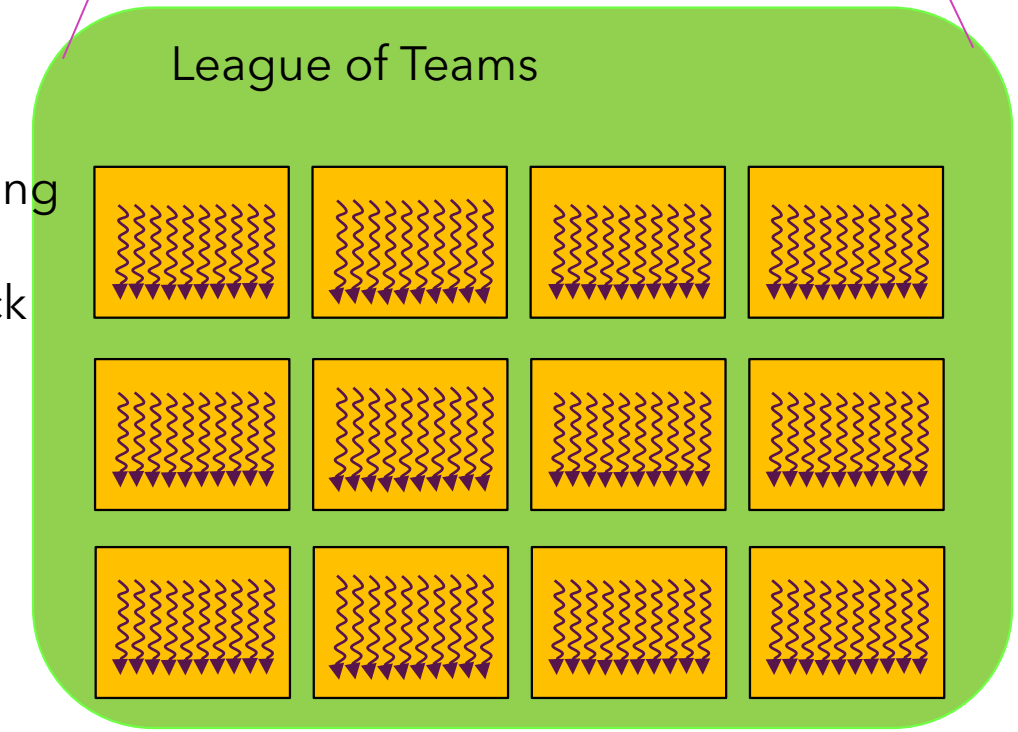
```
#pragma omp target teams
#pragma omp parallel
{
  // code block
}
```

or

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

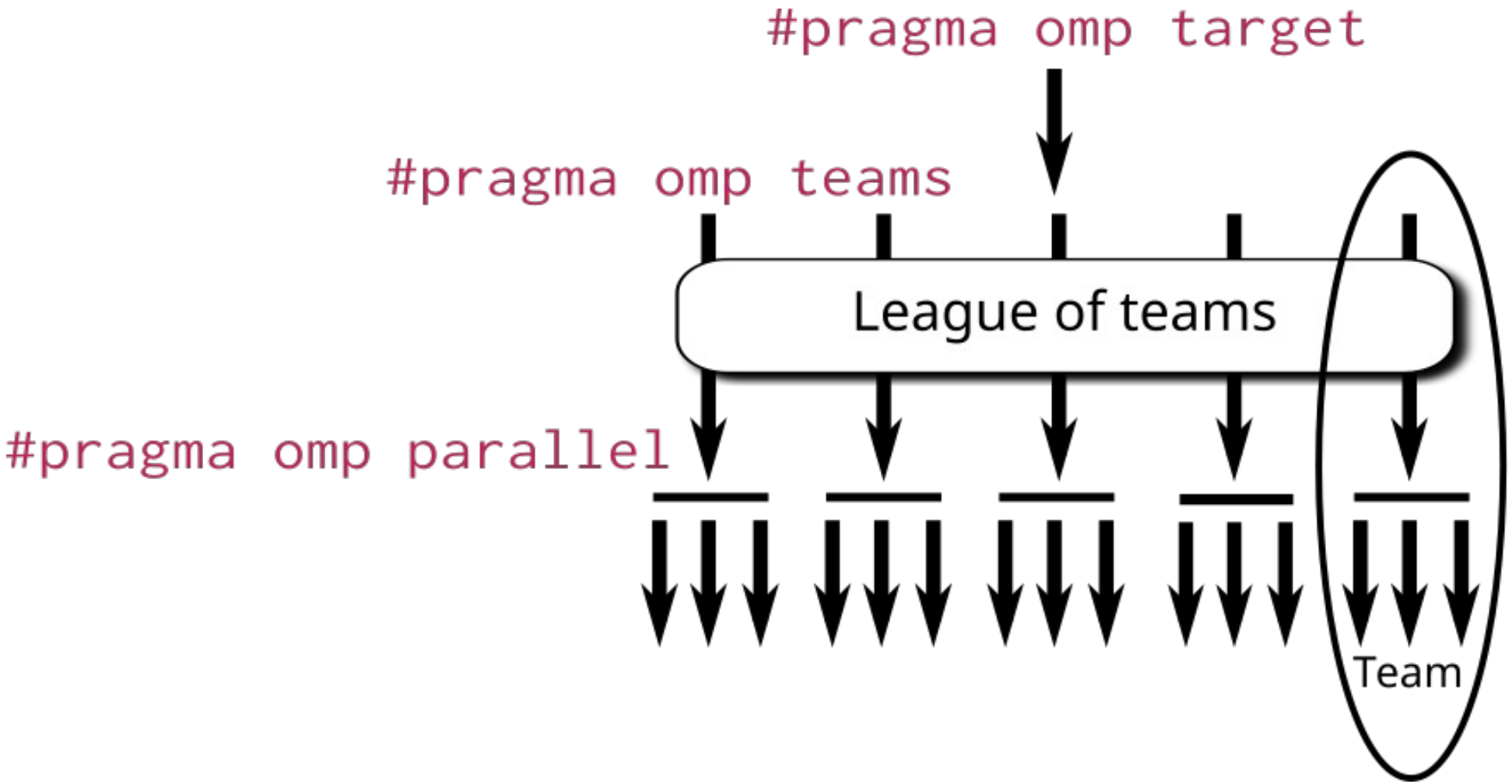


all executing the same code block



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

League of multi-threaded teams



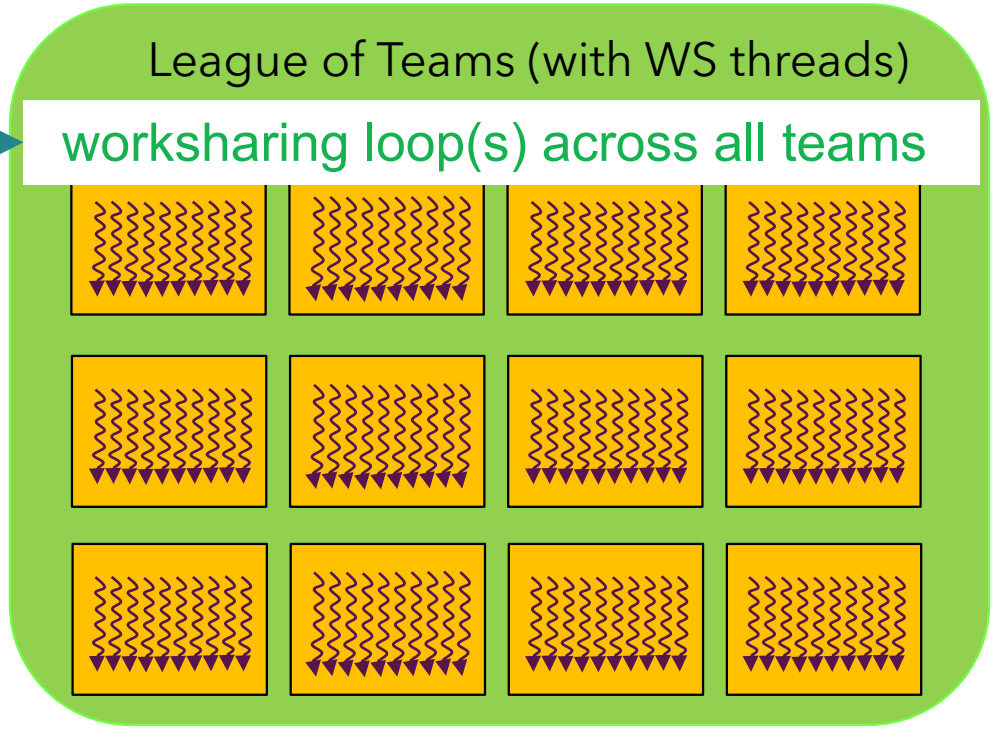
Distribute and parallel for Constructs

- distribute construct specifies iterations of a loop (of canonical form) are distributed across the initial threads of all teams
- iteration space is divided into chunks that are approximately equal in size
- distribute is combined with parallel for to create worksharing across all teams – all the SM/CUs.

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

or

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



Distribute and parallel for worksharing example

- distribute outer loop across teams, and workshare in loop in each team

```c
#pragma omp target
#pragma omp teams
#pragma omp distribute
for (int i = 0; i < N; i++)
  #pragma omp parallel
  #pragma omp for
  for (int j = 0; j < M; j++) {
    ...
  }
```

```fortran
!$omp target
!$omp teams
!$omp distribute
do i = 1, N
  !$omp parallel
  !$omp do
  do j = 1, N
    ...
  end do
  !$omp end do
  !$omp end parallel
end do
!$omp end distribute
!$omp end teams
!$omp end target
```

Controlling the number of teams and threads

- By default, the number of teams and the number of threads is up to the implementation to decide – it is reasonable to do so.
- Use num_teams and num_threads clauses for teams and parallel constructs
  - may improve performance in some cases
  - performance is most likely not portable

```
#pragma omp target
#pragma omp teams num_teams(32)
#pragma omp parallel num_threads(128)
{
  // code executed in device
}
```

OpenMP Implementation-Defined Behaviors

5.0
The number of teams created is implementation defined, but less than or equal to the value of the **num_teams** clause.

The number of threads that participate in each team is implementation defined but,less than or equal to the value of the **thread_limit** clause.

Loop Construct

- In OpenMP 5.0 a new **loop** worksharing construct has been introduced

- Leaves more freedom to the implementation to do the work division
  - It tells the compiler/runtime only that the loop iterations are independent and can be executed in parallel

```
#pragma omp target
#pragma omp loop
for (int i = 0; i < N; i++) {
  p[i] = v1[i] * v2[i]
}
```

```
!$omp target
!$omp loop
do i = 1, N
  p(i) = v1(i) * v2(i)
end do
!$omp end loop
!$omp end target
```

Compiler diagnostics

- Compiler diagnostics are usually the first thing to check when starting to work with OpenMP, as it can tell you:
  - what operations were actually performed
  - what kind of data copies were made
  - if and how the loops were parallelized

- Diagnostics are very compiler dependent
  - compiler flags
  - level and formatting of information

Cray Compiler diagnostics

- Different options (& behavior) for Cray C/C++ and Fortran

```
$ cc -fopenmp  -fsave-loopmark
```

```
$ ftn -fopenmp -hmsgs -hlist=m
```

```
ftn-6405 ft: ACCEL VECTORSUM, File = sum. F90, Line = 17

A region starting at line 17 and ending at line 21 was placed on the accelerator.

ftn-6823 ft: THREAD VECTORSUM, File sum. F90, Line = 17

A region starting at line 17 and ending at line 21 was multi-threaded.
```

Cray run time diagnostics

- Finding points of failure in code: use –g to get line number
- Use CRAY_ACC_DEBUG=# for better diagnostics.

```
$ CC -fopenmp -g vsum.cpp

$ env CRAY_ACC_DEBUG=2 ./a.out
```

Clang run time diagnostics

- Finding points of failure in code:
  - use -g to get line number (and symbols)
  - use -gline-tables-only (lacks all the scope/variable/type)
- Used LIBOMPTARGET_INFO=# for better diagnostics.

```
$ clang++ -fopenmp -g vsum.cpp

$ env LIBOMPTARGET_INFO=-1 ./a.out
```

target syntax

#pragma omp target [clauses]

!$omp target [clauses]
  structured-block
!$omp end target

```
if(logical-expr)
device(int-expr)
private(list)
firstprivate(list)
in_reduction(red-id: list)
map([map-type-modifier,...] map-type:] list)
is_device_ptr(list)
defaultmap(implicit-behavior[:var-category])
nowait
depend([dep-modifier,] dep-type: list)
allocate([allocator:]list)
uses_allocators(...)
```

*abridged syntax

Summary

- OpenMP enables directive-based programming of accelerators with C/C++ and Fortran

- Host--device model
  - host offloads computations to the device

- Host and device may have separate memories
  - host controls copying into/from the device

- Key concepts:
  - league of teams
  - threads within a team
  - worksharing between teams and threads with distribute

Useful resources:

- HPE Cray Programming Environment Documentation:
  - https://cpe.ext.hpe.com/docs/

- 2022 ECP Community BoF Days
  - https://www.openmp.org/wp-content/uploads/2022_ECP_Community_BoF_Days-OpenMP_RoadMap_BoF.pdf