Advanced OpenMP

OpenMP target offloading

Accelerator support in OpenMP

- Not GPU specific
  - Not many other interesting devices at the moment, however
- Fully integrated into OpenMP for the CPU
- Introduced in OpenMP 4.0, with significant revisions/extensions in 4.5 and 5.0
- Similar to, but not the same as, OpenACC directives.
  - OpenACC is an alternative standard for offloading to GPUs
  - Developed before OpenMP 4.0
- Current, usable implementations of OpenMP for GPUs are: Cray, IBM, LLVM/clang, gcc

OpenMP device model

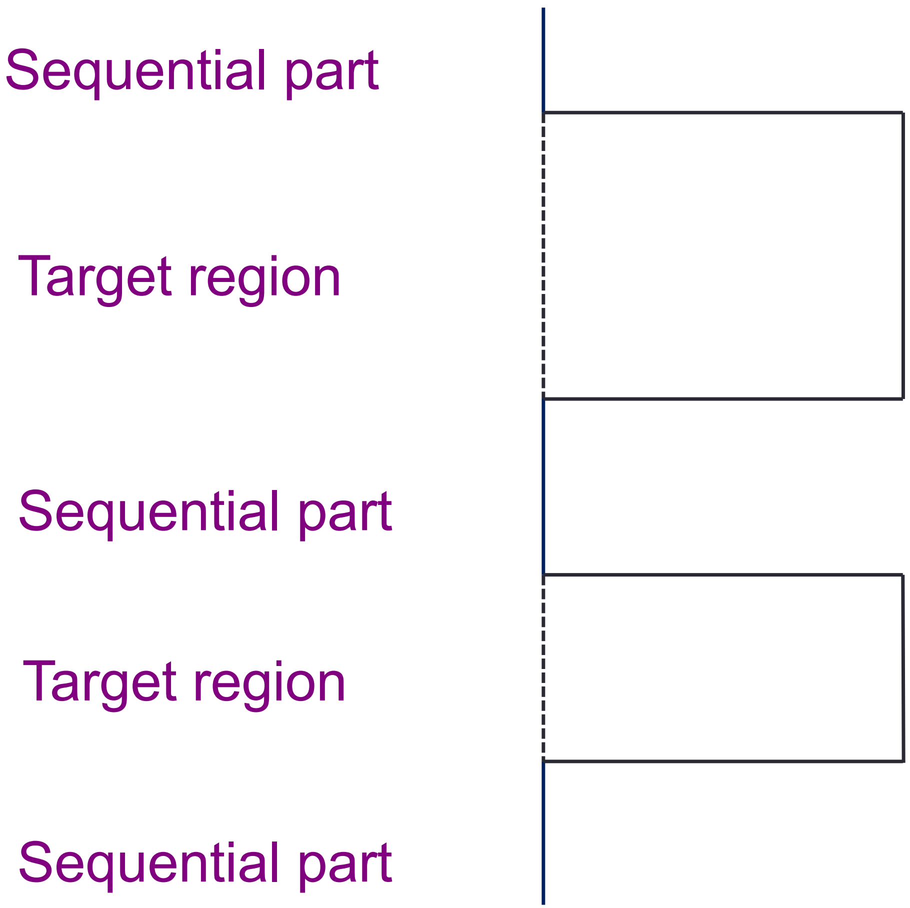
- Host-centric model with one host device and multiple target devices of the same type.
- *device*: a logical execution engine with local storage.
- *device data environment*: a data environment associated with a target data or target region.
- **`target`** constructs control how data and code is offloaded to a device.
- Data is mapped from a host data environment to a device data environment.

Target region

- The *target region* is the basic offloading construct in OpenMP.
- A target region defines a section of a program.
- The OpenMP program starts executing on the host
- When a target region is encountered, the code it contains is executed on a 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

```
#pragma omp target
    structured block
```

Target region




```
int main(){
.
.
#pragma omp target
{
.
.
.
.
.
.
.
}
.
.
.
.
#pragma omp target
{
.
.
.
}
.
.
.
```

Host and device data

- Host and device have separate memory spaces
- In order to access data inside the target region, it must be mapped to the device.
- Mapped data must not be accessed by the host until the target region has completed
- Default behaviour (in 4.5):
  - scalars referenced in the target construct are treated as *firstprivate* (new copy on device, initialised with value on host)
  - Static arrays are copied to the device on entry and back to the host on exit

Map clause

- More control is available via the map clause on the target construct

```
#pragma omp target map(map-type:list)
```

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 host on exit

`tofrom` copy the data to the device on entry and back on exit

`alloc` allocate an uninitialised copy on the device (don't copy values)

Example

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

- Sequential execution of the loop on the device – not very useful!

Dynamically allocated data

- Need to specify the number of elements to be copied:

```
int* B = (int*)malloc(sizeof(int)*N);
#pragma omp target map(to:B[0:N])
```

Can specify part of an array:

```
#pragma omp target map(to:B[10:3])
```

Note: syntax in C/C++ `[start:length]` is different from Fortran subarrays `[start:end]`!

Keeping data on the device

- Moving data between the host and device is expensive on a lot of current hardware
- Would like to avoid mapping data in every target region if it can be kept on the device between target regions
- `target data` constructs just map data and do not offload any code
- `target update` construct copies values between host and device between target constructs

Target data constructs

```
#pragma omp target enter data map(to: A[0:N],B[0:N])

for (r=0; r<reps; r++){
#pragma omp target
  {
    // do stuff with A and B
  }
  // do something on the host
}

#pragma omp target exit data map(from: B[0:N])
```

Target update construct

```
#pragma omp target enter data map(to: A[0:N],B[0:N])

#pragma omp target
  {
    // do stuff with A and B
  }
#pragma omp update from(A[0])
   // modify A[0] on the host
#pragma omp update to(A[0])
#pragma omp target
  {
    // do more stuff with A and B
  }
#pragma omp target exit data map(from: B[0:N])
```

Parallelism on the device

- In principle we can use all the ”normal” OpenMP constructs inside a target region to create and use threads on the device
  - Parallel regions, worksharing, synchronization, tasks, etc.
- However, GPUs are not able to support a full threading model outside of a single stream multiprocessor (SM)
  - no synchronization or memory fences possible between SMs
  - no coherency between L1 caches
  - a parallel region inside a target region will only execute on one SM
  - compare with CUDA – can only synchronise threads inside a thread block, not between thread blocks

Example

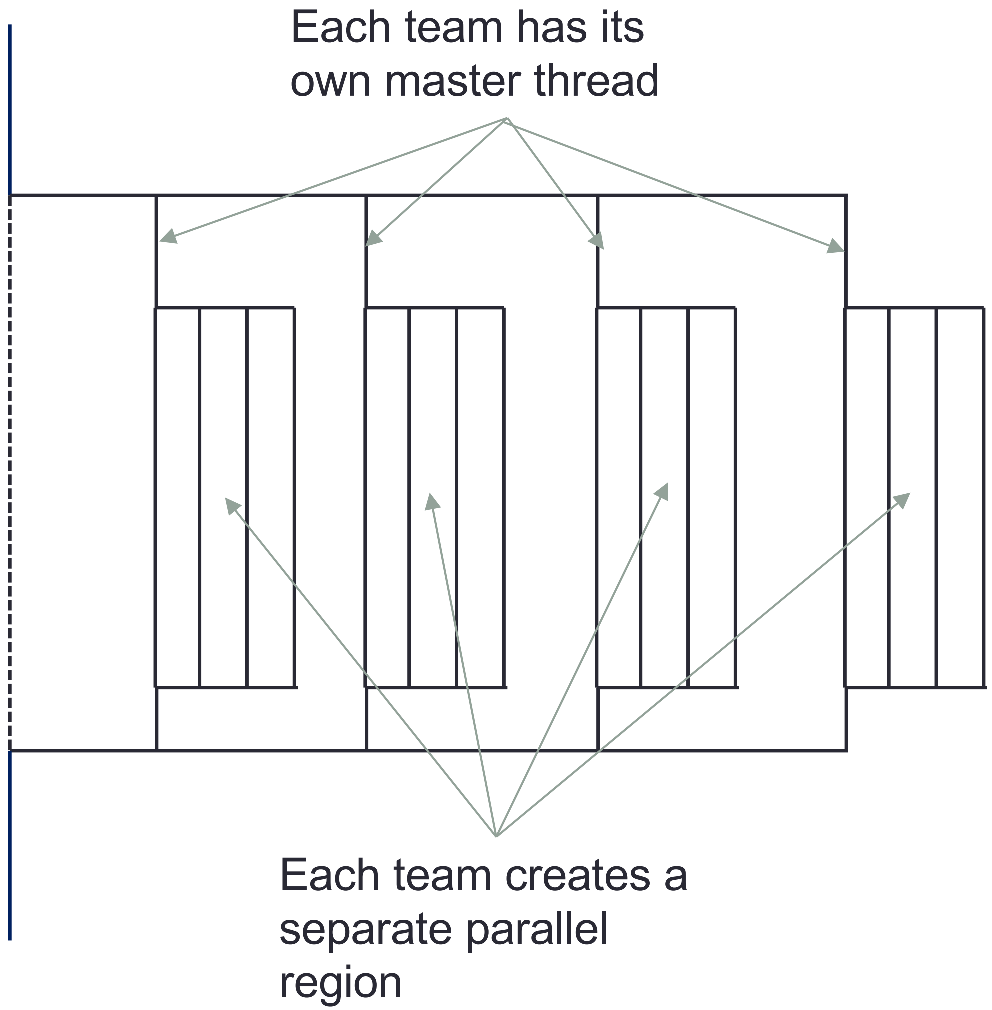
```
#pragma omp target map(to:B,C), map(tofrom:sum)
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; i++){
     sum += B[i] + C[i];
}
```

- On some devices, this might work fine
- On a GPU, this will only utilise one SM

Teams construct

- Creates multiple master threads inside a target region
- Each master thread can spawn its own team of threads with a parallel region
- Threads in different teams cannot synchronise with each other
  - Barriers, critical regions, locks, atomics only apply to the threads within a team
- Can set the number of teams, query the current team ID and query the number of teams

Teams and parallel regions




```
int main(){
.
.
#pragma omp target
#pragma omp teams
#pragma omp parallel
{
.
.
.
.
.
.
.
}
.
```

Distribute construct

- As ever, loops are the main source of parallelism in most applications
  - and especially so for GPUs
- If we offload a parallel loop to the device, we would like to distribute the iterations of the loop across the teams as well as across the threads within the teams
- **`distribute`** construct can be used to do this
- Like the **`for`** construct but, assigns iterations of the following loop to different teams
- Has a schedule clause **`dist_schedule`**, but the only schedule kind allowed is static, with a (optional) chunksize
-

Example

```
#pragma omp target teams distribute parallel for\
map(to:B,C), map(tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++){
      sum += B[i] + C[i];
}
```

- Distributes iterations across multiprocessors *and* across threads within each multiprocessor.
- Note the (long!) combined construct here

Calling functions inside target regions

- **`declare target`** compiles a version of function/subroutine that can be called on the device

```
#pragma omp declare target
int myfunc(int index);
#pragma omp end declare target

#pragma omp target teams distribute parallel for\
map(tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++){
       sum += myfunc(i);
}
```

Target directive clauses

- **`device`** clause allows the programmer to specify which device to offload to (if there is more than one).
  - Takes an integer parameter –device numbering is implementation dependent
- By default, the host thread blocks until target region is completed. Can change this behaviour with a **`nowait`** clause.
  - Target region is actually a task, so task synchronisation constructs (e.g. taskwait) can be used to wait for completion.
  - Need to make sure the host does not access mapped data until the target region completes

Performance issues

- Transferring data between host and device is expensive
- Need to minimize this as much as possible
  - Don't transfer anything that's not required
  - 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
  - For nested loops can use the `collapse` clause to parallelise two or more loops in the nest

Memory layout

- For CPUs having different threads accessing neighbouring words in memory can be bad
  - risk of false sharing
  - OK if its just reads
- For GPUs having different threads accessing neighbouring words in memory can be good
  - allows coalesced loads/stores
- If data structures are being used on both CPU and GPU, it may be best to explicitly change the layout (e.g. transpose multidimensional arrays) before mapping to the GPU.
  - Might also be possible to interchange loops, and/or use a `static,1` schedule

Reusing this material