OpenMP for Accelerators

an overview of the current proposal as of October 11th, 2012

Christian Terboven <terboven@rz.rwth-aachen.de>
11.10.2012 / Aachen, Germany
Stand: 10.10.2012
Version 2.3
History

- De-facto standard for Shared-Memory Parallelization.

- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.

- 08/2007: OpenMP 3.0 draft
- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release
- 11/2012: OpenMP 4.0 draft expected

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.
Some of the following information is changing daily!

- We expect the concepts to remain valid …
- … but i.e. the spelling seems to change quite quickly.
Agenda

- What is an Accelerator?
- Execution Model
- Data Model

The Target Construct
- Target Parallel
- Target Task
- Target Taskwait
- Target Data
- Target Flush
- Target Declare

Most content on these slides has been developed by James Beyer (Cray) and Eric Stotzer (TI), the leaders of the OpenMP for Accelerators subcommittee.
What is an Accelerator?
What kind of devices shall be supported?

- In how differs an accelerator from just another core?
  - different functionality, i.e. optimized for something special
  - additional instructions
  - different (possibly limited) instruction set
  → heterogeneous device

- Assumptions used as design goals for OpenMP 4.0:
  - an accelerator is attached to one or more host processors
  - it is probably heterogeneous
  - it may not be programmable in the same language as the host, or it may not implement all operations available on the host
  - almost always specialized for loop nests
Execution Model
Target constructs create device tasks that are executed by devices. Devices have device-only threads.

There is an initial device thread that is waiting to execute device tasks.

Device threads can spawn teams of device threads.

*The target construct modifies the subsequent OpenMP construct.*

Data clauses optimize the construction/deconstruction of the data environment.

```c
sum = 0;
#pragma omp target device(acc0) in(B,C)
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
    sum += B[i] * C[i]
```
Data Model
A device has a **data environment**

Variables are copied in and out of a *data environment*, or shared with the host – an implementation defines what it does

**Shared copy semantics:**

- Devices have memory that is shared with the host or private, local, memory
- Variables are either 'copied' to the private copy or are shared with the host processor
- For an implementation that supports multiple target types it is possible to support both

**Shared copies are synchronized either implicitly at the end of target construct regions or explicitly using a target flush construct**

**Note:** OpenMP 4.0 will bring shaping expressions to describe a slice of an array, the shape of a pointer, etc…
Possible problems of a Data Model

```c
#pragma omp data copyin(A)  // A on acc
{
    // A is now ‘defined’ on the accelerator
    // A is now ‘undefined’ on the host?
    // accelerator modifies A
}
// A is now undefined on the host and the accelerator?
// There is no copyout to make the host memory consistent?!?
```

- Shared copies are synchronized either implicitly at the end of target construct regions or explicitly using a target flush construct to avoid exactly these problems
Constructs
Create a device task that includes the structured block of the subsequent parallel construct. The encountering thread waits for the completion of the device task. The device task executes as an accelerated region (multiple threads).

C/C++

```cpp
#pragma omp target [clause [[, clause],...]]
#pragma omp parallel-construct
    structured-block
```

**Clauses**

- `device(device-descriptor)`
- `num_threads(integer-list)`
- `in(list), out(list), inout(list)`
- `devptr(list)`
- `if(condition)`

The `num_threads` clause takes a list of positive integers that specifies the number of threads at each level of nested parallelism.
/* accelerator region */
#pragma omp target device(acc0)
#pragma omp parallel for
{
    for (i=0; i<N; i++) ...
}

#pragma omp target num_threads(8, 4, 2)
#pragma omp parallel for // use num threads 8 to work-share loop
    for ( k = 0; k < NUM_K; k++ ) {
        #pragma omp parallel loop // use num threads 4 to work-share loop
            for ( j = 0; j < NUM_J; j++ ) {
                #pragma omp parallel for // use num threads 2 to work-share loop
                    for( i=0; i<N; i++ ) {
                        ...
                    }
    }
}
Create a device task that includes the structured block of the subsequent task construct. The encountering thread does not wait for the completion of the device task. The device task executes as a dispatch region (a single thread).

**C/C++**

```c
#pragma omp target [clause [[,] clause],...] newline
#pragma omp task-construct
   structured-block
```

**Clauses**

- `device(device-descriptor)`
- `num_threads(integer-list)`
- `in(list)`, `out(list)`, `inout(list)`
- `devptr(list)`
- `if(condition)`
Target Task Example

/* dispatch region */
#pragma omp target device(accl)
#pragma omp task
{
    G();
}

#pragma omp target
#pragma omp taskwait
Wait on the completion of the child device tasks of the current task.

C/C++

```c
#pragma omp target [clause [], clause, ...] newline
#pragma omp taskwait
```

**Clauses**

device(device-descriptor)

if(condition)
Create a device data task with no associated code and a data environment that includes the variables in the data clauses of the construct. The device data task is a parent task of all device tasks encountered in the region.

The encountering thread continues to execute the region in the context of a new implicit task. The device data task is not a child task of this implicit task.

**C/C++**

```c
#pragma omp target clause [ [, ,] clause],... ] newline
#pragma omp data
  structured-block
```

**Clauses**

- `device(device-descriptor)`
- `in(list), out(list), inout(list)`
- `devptr(list)`
- `if(condition)`
Flush (in) a variable from the data environment of the current task into the enclosing device data task, or flush (out) a variable from the enclosing device data task into the current task.

**C/C++**

```c
#pragma omp target clause [][,] clause],...] newline
#pragma omp flush
```

**Clauses**

- `device (device-descriptor)`
- `in (list)`
- `out (list)`
- `if (condition)`
Allocate a copy of the variable or function in the device memory.

C/C++
#pragma omp target declare clause [[,] clause],... newl.
    variable-or-function-declaration

Clauses
device(device-descriptor)
Thank you for your attention.