Introduction to OpenMP Device Offload

Swaroop Pophale
Computer Scientist, CSMD
Outline

• Introduction to OpenMP
• History of OpenMP
• Recap of OpenMP Worksharing
• Introduction to OpenMP Offload
• Offload Steps
• Expressing parallelism
• Useful Runtime Routines
• Hands On
Introduction to OpenMP

It is an Application Program Interface (API) to allow programmers to develop threaded parallel codes on shared memory computational units.

• Directives are understood by OpenMP aware compilers (others are free to ignore)
• Generates parallel threaded code
  – Original thread becomes thread “0”
  – Share resources of the original thread (or rank)
  – Data-sharing attributes of variables can be specified based on usage patterns
In spring, 7 vendors and the DOE agree on the spelling of parallel loops and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released. cOMPUnity, the group of OpenMP users, is formed to enable researcher participation and organize workshops.

1.0

1997

Minor modifications

1.1

2.0

C/C++ v 1.0. First hybrid applications with MPI* and OpenMP appear.

2.5

The merge of Fortran and C/C++ specifications begins.

Unified Fortran and C/C++. Bigger than both individual specifications combined.

Incorporates task parallelism. The OpenMP memory model is defined and codified.

Supports offloading execution to accelerator and coprocessor devices, SIMD parallelism, and more. Expands OpenMP beyond traditional boundaries.

Supports min/max reductions in C/C++.

OpenMP supports taskloops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.

Supports: Memory Management API, Reverse Offload, Loop construct, Detached tasks, Custom Mappers, Tools API

Supports: loop transformation (tiling, ...) improved "omp loop", variant overloading, runtime variant selection*, compiler agnostic "built-in assume"

Specification clarifications

In 2016, OpenMP 4.5 was released, which supports task loops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.

In 2017, OpenMP 5.0 was released, which supports memory management API, reverse offload, loop construct, detached tasks, custom mappers, and tools API.

In 2018, OpenMP 5.1 was released, which adds loop transformation (tiling, ...) improved "omp loop", variant overloading, runtime variant selection, and compiler agnostic "built-in assume".

In 2019, OpenMP 5.2 was released, which includes specification clarifications.

Permanent ARB

Auxiliary ARB

Reference: 2021 Exascale Computing Project Virtual Annual Meeting April 12 – 16, 2021
Recap: OpenMP Worksharing

```c
#pragma omp parallel
```

All threads will execute the region

```c
#pragma omp parallel for
```

All threads will execute a part of the iterations

- Creates a team of OpenMP threads that execute the structured-block that follows
- Number of threads property is generally specified by OMP_NUM_THREADS env variable or num_threads clause (num_threads has precedence)
Recap: OpenMP Worksharing

**Serial**

```c
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**

```c
#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 sequentially (overwriting values of C)
- Total time = `time_for_single_iteration * N`

**Parallel Worksharing**

```c
#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 distribute iteration space (roughly N/4 per thread)
- Total time = `time_for_single_iteration * N/4`
Introduction: OpenMP Offload

• OpenMP offload constructs are a set of directives for C++ and Fortran that were introduced in OpenMP 4.0 and further enhanced in later versions.
Summit vs. Frontier

**Summit Node**
- TF: 42 TF (6x7 TF)
- HBM: 96 GB (6x16 GB)
- DRAM: 512 GB (2x16x16 GB)
- NET: 25 GB/s (2x12.5 GB/s)
- MM/s: 83

**Frontier Node**
- TF: 96 TF (16x6 TF)
- HBM: 112 GB (8x14 GB)
- DRAM: 1024 GB (8x128 GB)
- NET: 40 GB/s (4x10 GB/s)
- MM/s: 163

HBM/DRAM Bus (aggregate B/W) 2.1 GB/s Write 6.0 GB/s Read

NVLINK X-Bus (SMP) PCIe Gen4 EDR

EDR IB
OpenMP Offload: Steps

• **Identification** of compute kernels
  – CPU initiates kernel for execution on the device

• Expressing **parallelism** within the kernel

• Manage **data transfer** between CPU and Device
  – relevant data needs to be moved from host to device memory
  – kernel executes using device memory
  – relevant data needs to be moved from device to main memory
Step 1: Identification of Kernels to Offload

• Look for compute intensive code and that can benefit from parallel execution
  – Use performance analysis tools to find bottlenecks
• Track independent work units with well defined data accesses
• Keep an eye on platform specs
  – GPU memory is a precious resource
• Confirm via Profiling
  – Tools like rocprof and HPCToolkit

• More information regarding rocprof can be found at: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#optimization-and-profiling
• More information on HPCToolkit can be found at: http://hpctoolkit.org
How to Offload?

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp target [clause[ [,] clause] ... ] new-line structured-block</td>
<td>!$omp target [clause[ [,] clause] ... ] loosely/tightly-structured-block !$omp end target</td>
<td>The target construct offloads the enclosed code to the accelerator.</td>
</tr>
</tbody>
</table>

- A device data environment is created for the structured block
- The code region is mapped to the device and executed.
OpenMP Offload: Target Directive

• Clauses allowed on the target directive:
  – `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)
  – `uses Allocators(allocator[[allocator-traits-array] [,allocator[[allocator-traits-array]] ...]])
OpenMP Offload: Example using `omp target`

/*C code to offload Matrix Addition Code to Device*/

```c
...
int A[N][N], B[N][N], C[N][N];
/* initialize arrays */
#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];
    }
  }
} // end target
```

The target construct is a task generating construct.
Step 2: Expressing Parallelism

/*C code to offload Matrix Addition Code to Device*/

... int A[N][N], B[N][N], C[N][N];
/* initialize arrays */
#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];
        }
    }
} // end target
Expressing Parallelism: Increasing device utilization

```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```

```
#pragma omp target teams
distribute parallel for
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```
## Expressing Parallelism: Device Execution Directives

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target [clause[ [,] clause] ... ] new-line structured-block</code></td>
<td><code>$omp target [clause[ [,] clause] ... ]</code> loosely/tightly-structured-block <code>$omp end target</code></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong>: The <code>target</code> construct offloads the enclosed code to the accelerator.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target teams [clause[ [,] clause] ... ] new-line structured-block</code></td>
<td><code>$omp target teams [clause[ [,] clause] ... ]</code> loosely/tightly-structured-block <code>$omp end target teams</code></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong>: The <code>target</code> construct offloads the enclosed code to the accelerator. The <code>teams</code> construct creates a league of teams. The initial thread of each team executes the code region.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target teams distribute [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>$omp target teams distribute [clause[ [,] clause] ... ]</code> loop-nest <code>$omp end target teams distribute</code></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong>: The <code>target</code> construct offloads the enclosed code to the accelerator. A league of thread teams is created, and loop iterations are distributed and executed by the initial teams.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target teams distribute parallel for [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>$omp target teams distribute parallel do [clause[ [,] clause] ... ]</code> loop-nest <code>$omp end target teams distribute parallel do</code></td>
<td></td>
</tr>
</tbody>
</table>
| **Description**: The `target` construct offloads the enclosed code to the accelerator. A league of thread teams are created, and loop iterations are distributed and executed in parallel by all threads of the teams. **Note**: *need simd to map to threads*
# Expressing Parallelism: Other combinations

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target parallel [clause[ [,] clause] ... ] new-line structured-block</code></td>
<td><code>!$omp target parallel [clause[ [,] clause] ... ]</code> <code>loosely-structured-block</code> <code>!$omp end target parallel</code></td>
<td>The <strong>target</strong> construct offloads the enclosed code to the accelerator. The <strong>parallel</strong> construct creates a team of OpenMP threads that execute the region.</td>
</tr>
<tr>
<td><code>#pragma omp target parallel for [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target parallel do [clause[ [,] clause] ... ]</code> <code>loop-nest</code> <code>!$omp end target parallel do</code></td>
<td>The <strong>target</strong> construct offloads the enclosed code to the accelerator. The <strong>parallel for/do</strong> combined construct creates a thread team and distributes the inner loop iterations over threads.</td>
</tr>
<tr>
<td><code>#pragma omp target parallel loop [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target parallel loop [clause[ [,] clause] ... ]</code> <code>loop-nest</code> <code>!$omp end target parallel loop</code></td>
<td>The <strong>target</strong> construct offloads the enclosed code to the accelerator. The <strong>parallel</strong> construct creates a team of OpenMP threads that execute the region. The <strong>loop</strong> construct allows concurrent execution of the associated loops.</td>
</tr>
<tr>
<td><code>#pragma omp target teams loop [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target teams loop [clause[ [,] clause] ... ]</code> <code>loop-nest</code> <code>!$omp end target teams loop</code></td>
<td>The <strong>target</strong> construct offloads the enclosed code to the accelerator. The <strong>teams</strong> construct creates a league of teams. The <strong>loop</strong> construct allows concurrent execution of the associated loops.</td>
</tr>
</tbody>
</table>
### Expressing Parallelism: SIMD

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp target simd [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target simd [clause[ [,] clause] ... ] loop-nest !$omp end target simd]</code></td>
<td>Semantics are identical to explicitly specifying a target directive immediately followed by SIMD directive.</td>
</tr>
<tr>
<td><code>#pragma omp target parallel for simd \ [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target parallel do simd [clause[ [,] clause] ... ] loop-nest !$omp end target parallel do simd]</code></td>
<td>Semantics are identical to explicitly specifying a target directive immediately followed by a parallel worksharing-loop SIMD directive.</td>
</tr>
<tr>
<td><code>#pragma omp target teams distribute simd \ [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target teams distribute simd [clause[ [,] clause] ... ] loop-nest !$omp end target teams distribute simd]</code></td>
<td>Semantics are identical to explicitly specifying a target directive immediately followed by a teams distribute simd directive.</td>
</tr>
<tr>
<td><code>#pragma omp target teams distribute parallel for simd \ [clause[ [,] clause] ... ] new-line loop-nest</code></td>
<td><code>!$omp target teams distribute parallel do simd [clause[ [,] clause] ... ] loop-nest !$omp end target teams distribute parallel do simd]</code></td>
<td>Semantics are identical to explicitly specifying a target directive immediately followed by a teams distribute parallel worksharing-loop SIMD directive.</td>
</tr>
</tbody>
</table>

*We will revisit this when we discuss Frontier specifics*
# Useful RT Routines: Device Environment

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Where to call?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int omp_get_num_procs(void);</td>
<td>integer function omp_get_num_procs()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>returns the number of processors available to the device</td>
</tr>
<tr>
<td>void omp_set_default_device(int device_num);</td>
<td>subroutine omp_set_default_device(device_num) integer device_num</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>sets the value of the default-device-var ICV of the current task to device_num</td>
</tr>
<tr>
<td>int omp_get_default_device(void);</td>
<td>integer function omp_get_default_device()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>returns the default target device</td>
</tr>
<tr>
<td>int omp_get_num_devices(void);</td>
<td>integer function omp_get_num_devices()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>returns the number of non-host devices available for offloading code or data.</td>
</tr>
<tr>
<td>int omp_get_device_num(void);</td>
<td>integer function omp_get_device_num()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>returns the device number of the device on which the calling thread is executing</td>
</tr>
<tr>
<td>int omp_is_initial_device(void);</td>
<td>logical function omp_is_initial_device()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>returns true if the current task is executing on the host otherwise, it returns false.</td>
</tr>
<tr>
<td>int omp_get_initial_device(void);</td>
<td>integer function omp_get_initial_device()</td>
<td><img src="https://example.com" alt="Host" /> <img src="https://example.com" alt="Target" /></td>
<td>return the device number of the host device</td>
</tr>
</tbody>
</table>
# Teams Region: Useful RT Routines

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
<th>Where to call?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int omp_get_num_teams(void);</td>
<td>integer function omp_get_num_teams()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>int omp_get_team_num(void);</td>
<td>integer function omp_get_team_num()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>void omp_set_num_teams(int num_teams);</td>
<td>subroutine omp_set_num_teams(num_teams) integer num_teams</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>int omp_get_max_teams(void);</td>
<td>integer function omp_get_max_teams()</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>void omp_set_teams_thread_limit(int thread_limit);</td>
<td>subroutine omp_set_teams_thread_limit(thread_limit) integer thread_limit</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
References

• Examples were adapted from: https://github.com/SOLLVE/sollve_vv

• OpenMP Specification (5.x)
  – https://www.openmp.org/specifications/

• https://www.nas.nasa.gov/hecc/assets/pdf/training/OpenMP4.5_3-20-19.pdf

• OpenMP Discussion @ 2021 Exascale Computing Project Virtual Annual Meeting (April 12 – 16, 2021)