GPU programming made easy with OpenMP

Aditya Nitsure (anitsure@in.ibm.com)
Pidad D'Souza (pidsouza@in.ibm.com)

OpenMP (Open Multi-Processing)

(https://www.openmp.org)

Take away

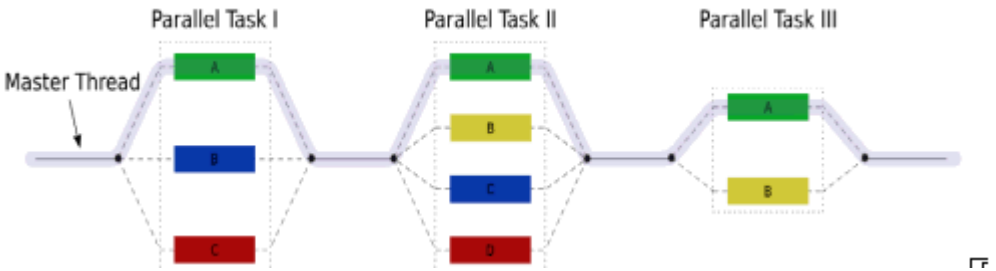
- What is OpenMP?
- How OpenMP programming model works ?
- How to write, compile and run OpenMP program?
- What are various OpenMP directives, runtime library API & Environment variables?
- How OpenMP program executes on accelerators (GPUs)?
- What are the best practices to write OpenMP program?

Introduction (1)

- Governed by OpenMP Architecture Review Board (ARB)
- Open source, simple and up to date with the latest hardware development
- Specification for shared memory parallel programming model for Fortran and C/C++ programming languages
  - compiler directives
  - library routines
  - environment variables
- Widely accepted by community – academician & industry

OpenMP Programming (1)

- The OpenMP API uses the fork-join model of parallel execution.
- The OpenMP API provides a relaxed-consistency, shared-memory model.
- An OpenMP program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially.
- When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team.



OpenMP Programming (2)

- Compiler directives
  -
- Runtime library routines
  -
- Environment variables
  - Todo… more details to be added

OpenMP Hello World

helloWorld.c

```
#include <stdio.h>
#include <omp.h>

int main()
{
   int i = 0;
   int numThreads = 0;
   // call to OpenMP runtime library
   numThreads = omp_get_num_threads(numThreads);

   // OpenMP directive
   #pragma omp parallel
   {
      // call to OpenMP runtime library
      int threadNum = omp_get_thread_num();
      printf("Hello World from thread %d \n", threadNum);
   }
   return 0;
}
```

```
// Setting OpenMP environment variable
export OMP_NUM_THREADS=4

//compile helloWorld program
xlc -qsmp=omp helloWorld.c -o helloWorld

// Run helloWorld program
./helloWorld
```

Output

```
[aditya@hpcwsw7 openmp_tutorial]$ ./helloWorld
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2

[aditya@hpcwsw7 openmp_tutorial]$ ./helloWorld
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
```

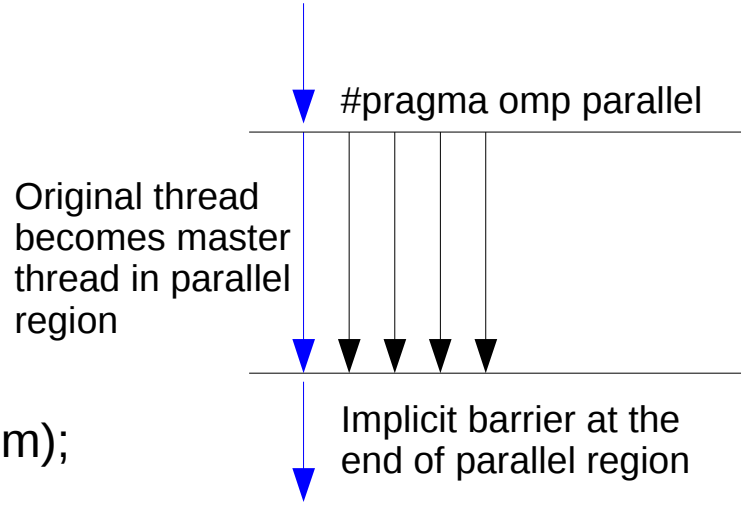
Compilation of OpenMP program

OpenMP Directives

- Parallel construct
- SIMD construct
- Combined construct
- Work sharing construct
- Master and synchronization construct
- Tasking construct
- Device construct

Parallel construct

#pragma omp parallel [clause[[,] clause] ...] new-line
{
}




```
#pragma omp parallel
{
      int threadNum = omp_get_thread_num();
      printf("Hello World from thread %d \n", threadNum);
}
```

- The fundamental construct that starts parallel execution
- A team of threads is created to execute parallel region
- Original thread becomes *master* of the new team
- All threads in the team executes parallel region

SIMD construct

More slides to follow on key directives

Combined construct

- Combination of more than one construct
  - Specifies one construct immediately nested inside another construct
- Clauses from both constructs are permitted
  - With some exceptions e.g. **nowait** clause cannot be specified in **parallel for** or **parallel sections**
- Examples
  - #pragma omp parallel for
  - #pragma omp parallel for simd
  - #pragma omp parallel sections
  - #pragma omp target parallel for simd

GPU offloading using OpenMP

Device construct (1)

CPU (no threading)

```
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```

for-loop

CPU OpenMP

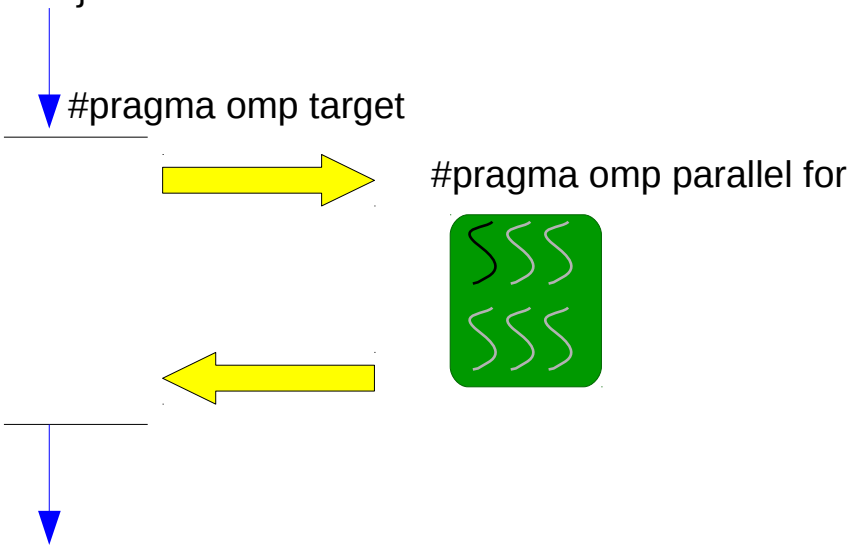
```
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```

#pragma omp parallel for

Implicit barrier

GPU OpenMP

```
#pragma omp target
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



Note

- No thread migration from one device to another.
- In absence of device or unsupported implementation for the target device, all target regions execute on the host.

Target construct

- Execute construct on device (GPU)
- Maps variables to device data environment

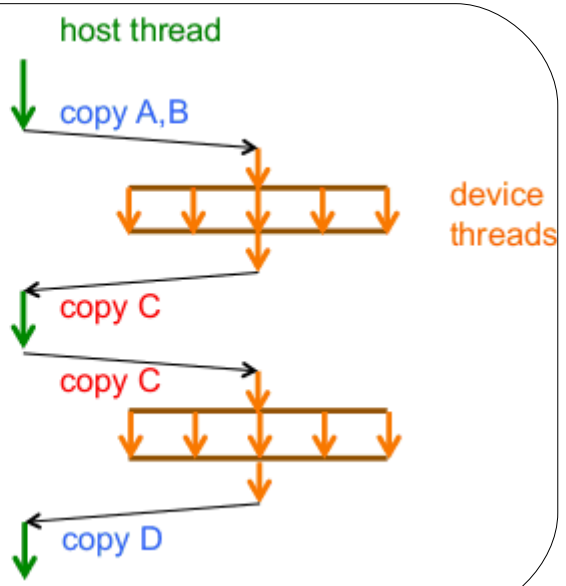
More charts to follow

Device construct (5)

```
double A[n,n], B[n,n], C[n,n];

#pragma omp target \
   map(to: A, B) map(from: C)
{
  // define C in terms of A, B
}

#pragma omp target \
   map(to: C) map(from: D)
{
  // define D in terms of C
}
```




```
real(8), dimension(:), allocatable :: A, B, C
allocate(A(N), B(N), C(N))
call init (A, B, C)

$!omp target enter data map (alloc: C)
$!omp target enter data map (to: A, B)

call foo (A, B, C)

$!omp target exit data map (delete: C, B)
$!omp target exit data map (from: A)
```

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target data map(alloc: C)
{
 #pragma omp target map(to: A, B) map(from: C)
 {
   // define C in terms of A, B
 }

 #pragma omp target map(from: D) map(to: C)
 {
   // define D in terms of C
 }
}
```

c is already in device scope

thus inner map clauses of c are ignored

OpenMP Runtime library (1)

- Runtime library definitions
  - omp.h stores prototype of all OpenMP routines/functions and type.
- Execution environment routines (35+ routines)
  - Routines that affects and monitor threads, processors and parallel environment
    - void omp_set_num_threads(int num_threads);
    - int omp_get_thread_num(void);
    - int omp_get_num_procs(void);
    - void omp_set_schedule(omp_sched_t kind, int chunk_size);
- Timing routines
  - routines that support a portable wall clock timer
  - double omp_get_wtime(void);
  - double omp_get_wtick(void);

OpenMP Environment Variables

1	OMP_SCHEDULE	sets the run-sched-var ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
2	OMP_NUM_THREADS	sets the nthreads-var ICV that specifies the number of threads to use for parallel regions.
3	OMP_DYNAMIC	sets the dyn-var ICV that specifies the dynamic adjustment of threads to use for parallel regions
4	OMP_PROC_BIND	sets the bind-var ICV that controls the OpenMP thread affinity policy.
5	OMP_PLACES	sets the place-partition-var ICV that defines the OpenMP places that are available to the execution environment.
6	OMP_NESTED	sets the nest-var ICV that enables or disables nested parallelism.
7	OMP_STACKSIZE	sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation.
8	OMP_WAIT_POLICY	sets the wait-policy-var ICV that controls the desired behavior of waiting threads.
9	OMP_MAX_ACTIVE_LEVELS	sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.
10	OMP_THREAD_LIMIT	sets the thread-limit-var ICV that controls the maximum number of threads participating in a contention group.
11	OMP_CANCELLATION	sets the cancel-var ICV that enables or disables cancellation.
12	OMP_DISPLAY_ENV	instructs the runtime to display the OpenMP version number and the initial values of the ICVs, once, during initialization of the runtime.
13	OMP_DEFAULT_DEVICE	sets the default-device-var ICV that controls the default device number.
14	OMP_MAX_TASK_PRIORITY	sets the max-task-priority-var ICV that specifies the maximum value that can be specified in the priority clause of the task construct.

Best practices

- To do