Offloading to GPU with OpenMP

Johan Hellsvik

PDC Center for High Performance Computing, KTH Royal Institute of Technology, Sweden

Introduction to GPU, Stockholm/Zoom, 15 September, 2023

Presentation based on the ENCCS lesson OpenMP for GPU offloading

OpenMP

OpenMP is a Application Programming Interface (API) to write shared memory parallel applications.

OpenMP is based on compiler directives, runtime routines, and environment variables.

OpenMP is available for C, C++, and Fortran

Programming model

OpenMP implements fork-join parallelism. During program execution some parts are executed sequentially, and some parts are executed in parallel.

Executation starts with a primary thread. Parallelism is achieved by the primary thread launching a team of threads.

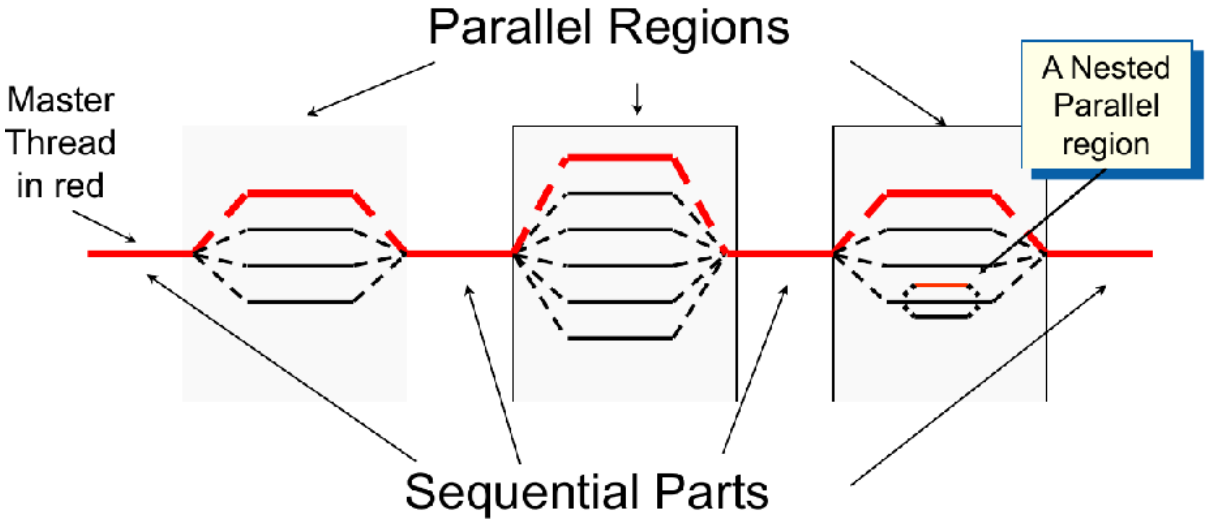

Figure from "Programming Your GPU with OpenMP", Deakin, McInstosh-Smith, Mattson, at SC22

OpenMP compiler directives

OpenMP compiler directives provide instructions for the compiler on how to compile the code.

If the compiler does not recognize the directives, serial code will be produced.

The OpenMP compilers directives have syntax

C/C++

```
#pragma omp directive clauses
```

Fortran

```
!$omp directive clauses
```

Hello world in Fortran

Print thread number and total number of threads

```
program hello_world_omp
  use omp_lib
  implicit none
  integer :: ithread, nthread
  !$omp parallel default(private)
  ithread=omp_get_thread_num()
  nthread=omp_get_num_threads()
  write(*,'(a16,i4,a16,i4,a16)') "This is thread ", ithread, &
       " out of a total of ",  nthread, " threads."
  !$omp end parallel
end program hello_world_omp
```

where `omp_get_thread_num()` and `omp_get_num_threads()` are OpenMP runtime routines.

Hello world in C

Print thread number and total number of threads

```
#include <omp.h>
#include <stdio.h>
void main()
{
    #pragma omp parallel
    {
    int ithread = omp_get_thread_num();
    int nthread = omp_get_num_threads();
    printf("This is thread %d out of a total of %d threads \n", ithread, nthread);
    }
}
```

where `omp_get_thread_num()` and `omp_get_num_threads()` are OpenMP runtime routines.

Shared memory, scoping clauses

The threads of an OpenMP program interact through read and writes to shared memory with common address space.

The access attributes of variables can be controlled by the clauses

- private
- shared
- default (none)

For example, in the directive

```
#pragma omp parallel for default(shared) private(a,b)
```

is specified that variables by default are shared, and that variables `a` and `b` are private.

Shared clause

The clause `default(shared)` sets the variable attributes to be shared. This can be augmented with the specification that some variables should be private.

```
int a, b, c, d
#pragma omp parallel for default(shared) private(a,b)
for (int i = 0; i < n; i++)
{
  // a and b are private variables
  // c and d are shared variables
  // the loop iteration variable i is private by default
}
```

Default(none)

The default(none) clause forces the programmer to explicitly specify the data-sharing attributes of all variables

The compiler will throw errors if the data sharing attribute is missing for one or more of the variables

```
int a, b, c, d
#pragma omp parallel default(none) private(a,b), shared(c,d)
for (int i = 0; i < n; i++)
{
  // a and b are private variables
  // c and d are shared variables
}
```

Example: A serial code that operates on arrays `vecA`, `vecB`, and `vecC`. How can this be parallelized by means OpenMP?

```
/* Copyright (c) 2019 CSC Training */
/* Copyright (c) 2021 ENCCS */
#include <stdio.h>
#include <math.h>
#define NX 102400

int main(void)
{
  double vecA[NX],vecB[NX],vecC[NX];
  double r=0.2;

/* Initialization of vectors */
  for (int i = 0; i < NX; i++) {
     vecA[i] = pow(r, i);
     vecB[i] = 1.0;
  }
```

```
/* Dot product of two vectors */
  for (int i = 0; i < NX; i++) {
     vecC[i] = vecA[i] * vecB[i];
  }

  double sum = 0.0;
  /* Calculate the sum */
  for (int i = 0; i < NX; i++) {
    sum += vecC[i];
  }
  printf("The sum is: %8.6f \n", sum);
  return 0;
}
```

OpenMP for heterogeneous systems

OpenMP which was originally developed and used for shared memory parallelization on homogenous systems.

Starting with version 4.0 OpenMP supports execution on heterogeneous systems. OpenMP uses the `TARGET` construct to offload execution from the host to the target device(s).

Device execution model

The execution on the device is host-centric

- i. the host creates the data environments on the device(s)
- ii. the host maps data to the device data environment
- iii. the host offloads OpenMP target regions to the target device to be executed
- iv. the host transfers data from the device to the host
- v. the host destroys the data environment on the device

Offloading to GPU with OpenMP

The OpenMP programming model can be used for directive based offloading to GPUs.

Example: A serial code that operates on arrays `vecA`, `vecB`, and `vecC`

```
/* Dot product of two vectors */
  for (int i = 0; i < NX; i++) {
     vecC[i] = vecA[i] * vecB[i];
  }
```

can be augmended to a parallel code by adding OpenMP directives

```
/* dot product of two vectors */
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < NX; i++) {
     vecC[i] = vecA[i] * vecB[i];
  }
```

TARGET construct

The `TARGET` construct is used to

- transfer control flow from the host to the device,
- transfer data between host and device.

If data is already existing on the device from earlier execution, no transfer of data will be made.

C/C++

```
#pragma omp target [clauses]
     structured-block
```

Fortran

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

TEAMS construct

OpenMP separates offload and parallelism

- The transfer of control to device with the TARGET construct is sequential and synchronous.
- Parallel regions on the device needs to be created explicitly
- The `TEAMS` construct creates a league of one-thread teams.
- The thread of each team execute concurrently and in its own contention group
- The `TEAMS` construct must be contained in a `TARGET` construct

C/C++

```
#pragma omp teams [clauses]
     structured-block
```

Fortran

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

DISTRIBUTE construct

The `DISTRIBUTE` construct is a coarsely worksharing construct

- Loop iterations are distributed across the master threads in teams
- No worksharing withing the threads in one team
- No implicit barrier at the end of construct.

C/C++

```
#pragma omp distribute [clauses]
     for-loops
```

Fortran

```
!$omp distribute [clauses]
      do-loops
!$omp end distribute
```

PARALLEL FOR/DO construct

The `PARALLEL FOR/DO` construct is used to create threads within each team and to distribute loop iterations across threads

C/C++

```
#pragma omp distribute [clauses]
     for-loops
```

Fortran

```
!$omp distribute [clauses]
      do-loops
!$omp end distribute
```

Comparison TEAMS DISTRIBUTE and PARALLEL FOR/DO

TEAMS DISTRIBUTE construct

- Coarse-grained parallelism
- Spawns multiple single-thread teams
- No synchronization of threads in different teams

PARALLEL FOR/DO construct

- Fine-grained parallelism
- Spawns many threads in one team
- Threads can synchronize in a team

Explicit and implicit data movement

Host and device have distinct memory spaces, wherefore transferring of data becomes inevitable.

A combination of both explicit and implicit data mapping is used.

If the variables are not explicitly mapped, the compiler will do it implicitly:

- A scalar is mapped as firstprivate. The variable is not copied back to the host
- Non-scalar variables are mapped with a map-type tofrom
- a C/C++ pointer is mapped as a zero-length array section
- Note that only the pointer value is mapped, but not the data it points to

Data mapping

The MAP clause on a device construct explicitly specifies how items are mapped from the host to the device data environment. The common mapped items consist of arrays(array sections), scalars, pointers, and structure elements.

`map(to:list)` On entering the region, variables in the list are initialized on the device using the original values from the host

`map(from:list)` At the end of the target region, the values from variables in the list are copied into the original variables on the host. On entering the region, the initial value of the variables on the device is not initialized

`map(tofrom:list)` the effect of both a map-to and a map-from

`map(alloc:list)` On entering the region, data is allocated and uninitialized on the device

`map(list)` equivalent to `map(tofrom:list)`

Optimizing Data Transfers

The performance of OpenMP offloading to device can be enhanced by

- Explicit mapping of data instead of implicit mapping.
- Reducing the amount of data transfer between host and device.
- Trying to keep the data environment residing on device as long as possible.

Data regions

Structured Data Region

The `TARGET DATA` construct is used to create a structured data region which is convenient for providing persistent data on the device which could be used for subseqent target constructs.

- Start and end points within a single subroutine
- Memory exists within the data region

C/C++

```
#pragma omp target data clause [clauses]
      structured-block
```

Fortran

```
!$omp target data clause [clauses]
        structured-block
!$omp end target data
```

Unstructured Data Region

- Multiple start and end points across different subroutines
- Memory exists until explicitly deallocated

C/C++

```
#pragma omp target enter data [clauses]
   // Code
#pragma omp target exit data [clauses]
```

Fortran

```
!$omp target enter data [clauses]
   ! code
!$omp target exit data [clauses]
```

Exercise 1: Dot product with OpenMP

Build and test run a Fortran program that calculates the dot product of vectors.

```
! Copyright (c) 2019 CSC Training
! Copyright (c) 2021 ENCCS
program dotproduct
  implicit none

  integer, parameter :: nx = 102400
  real, parameter :: r=0.2

  real, dimension(nx) :: vecA,vecB,vecC
  real    :: sum
  integer :: i

  ! Initialization of vectors
  do i = 1, nx
     vecA(i) = r**(i-1)
     vecB(i) = 1.0
  end do
```

```
  ! Dot product of two vectors
   !$omp target teams distribute map(from:vecC) &
   map(to:vecA,vecB)
   do i = 1, nx
      vecC(i) =  vecA(i) * vecB(i)
   end do
   !$omp end target teams distribute

   sum = 0.0
   ! Calculate the sum
   !$omp target map(tofrom:sum)
   do i = 1, nx
      sum =  vecC(i) + sum
   end do
   !$omp end target
   write(*,*) 'The sum is: ', sum

end program dotproduct
```

- Activate the PrgEnv-cray environment `ml PrgEnv-cray`
- Download the source code F90 or C

```
wget https://github.com/ENCCS/openmp-gpu/raw/main/content/exercise/ex04/solution/ex04.F90
wget https://github.com/ENCCS/openmp-gpu/raw/main/content/exercise/ex04/solution/ex04.c
```

- Load the ROCm module and set the accelerator target to amd-gfx90a

  `ml rocm/5.0.2 craype-accel-amd-gfx90a`
- Activate verbose runtime information

  `export CRAY_ACC_DEBUG=3`
- Compile the code on the login node

```
ftn -fopenmp ex04.F90 -o ex04_f90.x
cc -fopenmp ex04.c -o ex04.x
```

Run the code as a batch job

- Edit job_gpu_ex04.sh to specify the compute project and reservation

```
#!/bin/bash
#SBATCH -A edu23.introgpu    # Set the allocation to be charged for this job
#SBATCH -J myjob             # Name of the job
#SBATCH -p gpu               # The partition
#SBATCH -t 01:00:00          # 1 hour wall-clock time
#SBATCH --nodes=1            # Number of nodes
#SBATCH --ntasks-per-node=1  # Number of MPI processes per node

ml rocm/5.0.2                # Load a ROCm module
ml craype-accel-amd-gfx90a   # set the accelerator target

srun ./ex04.x > output.txt   # Run the ex04.x executable
```

```
#SBATCH --reservation=<name of reservation>
```

- Submit the script with `sbatch job_gpu_ex04.sh`
- with program output `The sum is:  1.25` written to `output.txt`

Optionally, test the code in interactive session.

- First queue to get one GPU node reserved for 10 minutes
  - `salloc -N 1 -t 0:10:00 -A <project name> -p gpu` where for the current course
  - `salloc -N 1 -t 0:10:00 -A edu23.introgpu -p gpu`
- wait for a node, then run the program `srun -n 1 ./ex04.x`
- with program output to standard out `The sum is:  1.25`

- Alternatively, login to the reserved GPU node (here nid002792) `ssh nid002792`.
- Load ROCm, activate verbose runtime information, and run the program
  - `ml rocm/5.0.2`
  - `export CRAY_ACC_DEBUG=3`
  - `./ex04.x`
- with program output to standard out

```
ACC: Version 5.0 of HIP already initialized, runtime version 50013601
ACC: Get Device 0
...
...
ACC: End transfer (to acc 0 bytes, to host 4 bytes)
ACC:
The sum is:  1.25
ACC: __tgt_unregister_lib
```

Exercise 2: Optimize data transfer

This is the Exercise: Data Movement in the ENCCS OpenMP for GPU offloading lesson

The exercise is about optimization and explicitly moving the data using the `target data` family constructs. The code for the exercise

Build and run the initial version of the code

- Clone the git repository and go to the directory of the data mapping exercise

```
git clone https://github.com/ENCCS/openmp-gpu.git
cd openmp-gpu/content/exercise/data_mapping
```

- Load the ROCm module and set the accelerator target to amd-gfx90a. Activate the PrgEnv-gnu environment. Using the Makefile, compile the code on the login node. Specify use of the Gnu compilers.

```
ml rocm/5.0.2 craype-accel-amd-gfx90a
ml PrgEnv-gnu
COMP=gnu make
```

- Run the executable `heat_serial` in a batch job

Add directives for data movement. Rebuild and run

- Three incomplete functions are added to explicitly move the data around in `core.cpp` or `core.F90`.
- You need to add the directives for data movement for them.
- Then recompile and run the code. A template solution can be found in `openmp-gpu/content/exercise/solution/data_mapping`.

Summary

OpenMP can be used for shared memory programming on

- Homogeneous hardware, as in OpenMP threading within CPU nodes
- Heterogeneous hardware, by means of offloading computation and data from host to device
- Can be combined with the Message Passing Interface (MPI) in order to create hybrid parallel code. See the ENCCS lesson episode Multiple GPU programming with MPI

Tools

HPE Cray CrayPat performance analysis tools

HPE Cray Code Parallelization Assistant

AMD rocgdb debugger

AMD rocprof and roctracer performance analysis tools

References

The official OpenMP website

NERSC Documentation on OpenMP

ENCCS lesson GPU Programming: When, Why and How?

ENCCS lesson OpenMP for GPU offloading

Programming Your GPU with OpenMP: A Hands-On Introduction (exercises and solutions repository)

HPE Compiler GPU offloading

OpenMP Tutorials at SC23