CS 610: Programming GPUs with OpenMP

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-I

Programming GPUs

Correctly and efficiently programming GPUs is challenging

- Different programming model compared to CPUs, arguably more **sophisticated** synchronization APIs, and requires **awareness** of the memory hierarchy for efficiency
- Development tools are less mature compared to CPU programming

Ways to program a GPU

(i) Use frameworks like Kokkos or AMReX to automate the parallelization

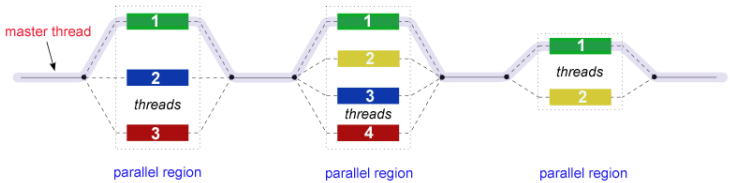
(ii) Identify compute-intensive accelerator-offloadable regions and use directive-based models like OpenMP or OpenACC

(iii) Natively program the GPU with CUDA, HIP, OpenCL, or SYCL

Shared Memory Programming with OpenMP

OpenMP is the de facto standard for thread-based shared memory parallelism

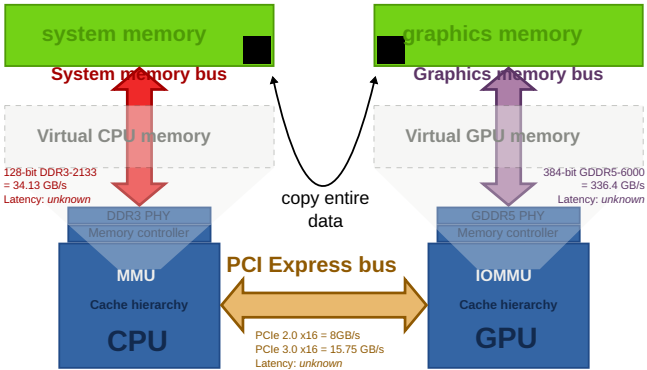
- Hints (directives) are used to parallelize relevant regions of code
- Goal is to ease parallelization of sequential programs




```
#include <omp.h>
int main() {
   // serial code, master thread
   ...
   // begin parallel section,
   // fork a team of threads
#pragma omp parallel ...
{
   // parallel region executed by
   // all threads
   ...
   // all parallel threads join
   // the master thread
}
   // resume serial code
   ...
}
```

OpenMP Offload

- By a heterogeneous setup, we mean a general-purpose processor connected to one or more accelerators (e.g., Intel Xeon processor connected with an NVIDIA GPU)
- OpenMP v4+ provides directives to support heterogeneous computing (e.g., GPU, TPU, DSP, and FPGA)



How to Offload Computation?

High level steps

(i) Identify compute-intensive code regions (i.e., kernels) that can benefit from data parallelism

(ii) Express the parallelism in the kernel

(iii) Manage data transfer between the host and the device

Execution model

- There is a single host device but there can be multiple target devices
- A device is a logical execution engine with its own local storage and data environment

Offload using the `target` directive

`#pragma omp target [clause ...]`

- When a host thread encounters the `target` directive, the region specified by it is executed by a new thread running on an accelerator, similar to a CUDA kernel
- The `target` construct offloads the enclosed code (control and data) and transfers control to the accelerator via a target task
- Transfer of control is synchronous, i.e., host thread waits for the offloaded computation to complete
- Offloaded code should be a data-parallel structured block that can be benefit from multiple threads on the accelerator

Offload Example Using GCC

Configure GCC to run OpenMP offload

- Check whether GCC was built with offload support with `gcc -v`
- The output should contain entries like `–enable-offload-targets=nvptx-none`
- You may need to install additional packages

```
sudo apt install gcc-12 g++-12 gcc-12-offload-nvptx nvptx-tools
```

- If not, then we will need to build from source with offload support

hello-world.cpp

Device Execution Directives

`#pragma omp target teams num_teams(16)`

- The `teams` construct creates a *league* of teams where each team is composed of a master thread and a number of worker threads
  - `num_teams` clause specifies the number of teams
  - The number of teams is implementation-dependent without a `num_teams` clause
- The **master** thread of each team **executes** the code region

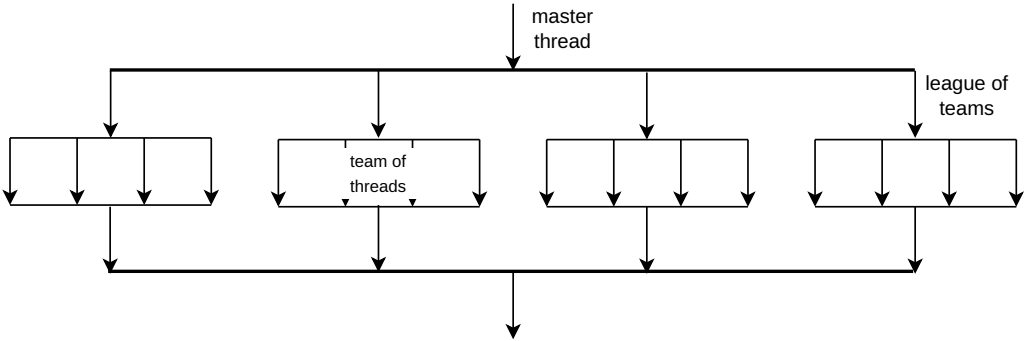
`#pragma omp target teams distribute`

Worksharing construct that **distributes** iterations of a loop among the master threads of the teams, so each master thread executes a **subset** of the iterations

Device Execution Directives

`#pragma omp target teams distribute parallel for`

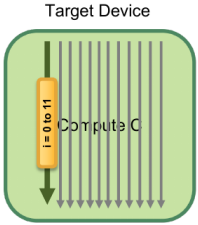
A league of thread teams are created, and loop iterations are distributed and executed in parallel by **all threads** of the teams



Expressing Parallelism: Increasing Device Utilization

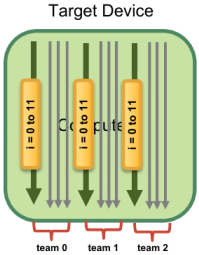
target

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



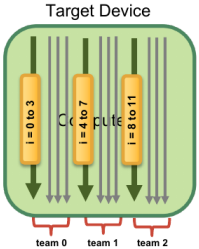
target teams

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



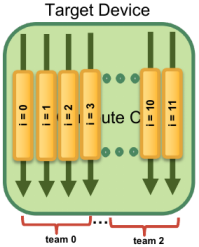
target teams distribute

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



target teams distribute parallel

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



Swaroop Pophale, Reuben Budiardja, and Wael Elwasif. Introduction to OpenMP Offload: Part 1.

Hierarchical Parallelization

OpenMP separates offload and parallelism

- Programmers need to explicitly create parallel regions on the target device

```
#pragma omp target teams distribute parallel for num_teams(4096)
    thread_limit(128) map(to : A[0 : N], B[0 : N]) map(from : C_gpu[0 : N])
  for (uint64_t i = 0; i < N; i++) {
    C_gpu[i] = A[i] + B[i];
  }
```

vector-addition.cpp

Summary of Device Execution Directives

```
#pragma omp target
```

- Offloads the enclosed code to the accelerator

```
#pragma omp target teams
```

- Creates a league of teams
- The master thread of each team executes the code region

```
#pragma omp target teams distribute
```

- Creates a league of thread teams
- Loop iterations are distributed and executed by the master threads in the teams

```
#pragma omp target teams distribute parallel for
```

- A league of thread teams are created
- Loop iterations are distributed and executed in parallel by all threads of the teams

Offloading using `target teams + loop`

```
#pragma omp target teams
{
// bind(teams) is implicit
#pragma omp loop
  for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B [i]
  }
}
```

```
void fun1() {
// Orphaned loop needs explicit binding
#pragma omp loop bind(teams)
  for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B [i]
  }
}
...
#pragma omp target teams
{
  fun1();
}
```

Summary of Device Execution Directives

```
#pragma omp target parallel
```

- Creates one team of OpenMP threads that execute the same region

```
#pragma omp target parallel for
```

- Creates one thread team and distributes the inner loop iterations over threads

```
#pragma omp target parallel loop
```

- Creates a team of OpenMP threads that execute the region
- Allows concurrent execution of the associated loops

```
#pragma omp target teams loop
```

- Allows concurrent execution of the associated loops by different teams

Comparing CUDA and OpenMP Terminologies

CUDA	OpenMP
`target`	Kernel launch
CUDA thread	OpenMP thread or SIMD lane
Thread block	Team
Thread block size	Team size
Number of thread blocks	Number of teams
Warp (size = 32)	SIMD chunk (`simdlen` = 8, 16, 32)
Maximum number of threads per block	Thread limit
`parallel` inside teams	Threads inside a block
`distribute`	splitting loop across blocks
`parallel for` inside `distribute`	splitting across threads

Mapping Data Between Host and Device

The `map` clause specifies how data items are mapped from the host to the device

`map(to:list)`	On entering the region, allocates memory on the device and initializes them using the values from the host
`map(from:list)`	Allocates memory on the device, and copies the values to the variables on the host on exiting the region.
`map(tofrom:list)`	Bidirectional copy
`map(alloc:list)`	On entering the region, data is allocated on the device
`map(delete:list)`	Free device memory

If a variable is not a scalar then it is treated as if it is mapped with a map-type of `tofrom`

Optimizing Data Transfers

```
// Maps variables to a device data environment for the extent of the region
#pragma omp target data map(to: A, B)
{
#pragma omp target map(from: C)
{
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      C[i][j] = A[i][j] + B[i][j];
} // end inner target
/* Some computation on host using C (no changes to A and B) */
#pragma omp target map(to: C) map(from: D)
{
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      D[i][j] = A[i][j] + B[i][j] C[i][j];
} // end inner target
} // end outer target
```

Asynchronous Offload

- OpenMP target constructs are synchronous by default
  - The host thread waits for the target region to end before continuing
- Try to overlap CPU and GPU computations for better performance
  - Use `nowait` clause if the host thread does not need to wait
  - Use `taskwait` to ensure that the host thread waits for the completion of dependent offloaded tasks

Overlap CPU and GPU Computation

```
1  int main() {
2    float a[N], b[N], c, d;
3    ...
4  #pragma omp target nowait map(to:b[0:N], c, d) map(from:a[0:N])
5  {
6    #pragma omp teams distribute parallel for
7    for (int i = 0; i < N; i++)
8      a[i] = b[i]*c + d;
9  }
10    func(b); // perform computation independent of device output
11    #pragma omp taskwait
12    func(a); // perform computation dependent of device output
13    ...
14 }
```

Asynchronous Offloads

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:a)
    init_data(a);
  #pragma omp target map(to:a[:N]) map(from:x[:N]) nowait depend(in:a) depend(
      out:x)
    compute_1(a, x, N);
  #pragma omp target map(to:b[:N]) map(from:z[:N]) nowait depend(in:b) depend(
      out:z)
    compute_3(b, z, N);
  #pragma omp target map(to:z[:N], x[:N]) map(from:y[:N]) nowait depend(in:z,x)
       depend(out:y)
    compute_4(z, x, y, N);
  #pragma omp taskwait
}
```

Code Examples

- README (.org file)
- hello-world.cpp
- vector-addition.cpp
- matmul.cpp
- compute-pi.cpp
- Makefile

References

- Swaroop Pophale, Reuben Budiardja, and Wael Elwasif. Introduction to OpenMP Offload: Part 1.
- Swaroop Pophale. Introduction to OpenMP Device Offload: Data Movement
- Michael Klemm. Intro to GPU Programming with the OpenMP API.
- EuroCC National Competence Center Sweden. OpenMP for GPU Offloading.
- Tom Deakin and Wei-Chen Lin. Programming Your GPU with OpenMP.
- Joesph Huber. OpenMP Offloading Features in LLVM 15.