

OpenMP for GPU

Kenjiro Taura

OpenMP for GPU

- recent OpenMP supports offloading to GPU (`target` directive)
- official home page: <http://openmp.org/>
- specification:
<http://openmp.org/wp/openmp-specifications/>
- latest version is 5.0
(<https://www.openmp.org/spec-html/5.0/openmp.html>)
- section numbers below refer to those in OpenMP spec 5.0

Compiling OpenMP programs for GPUs

- LLVM (clang/clang++) : compile with `-fopenmp`
`-fopenmp-targets=nvptx64`

```
1 $ clang -Wall -fopenmp -fopenmp-targets=nvptx64 program.c
```

- you get a warning: “CUDA version is newer than the latest supported version 11.5” and `-Wunknown-cuda-version` suppresses it

- NVIDIA HPC SDK (nvc/nvc++) : compile with `-mp`
`-target-gpu`

```
1 $ nvc -Wall -mp -target=gpu program.c
```

Directives overview

- ① move control
 - `target` : moves the execution to GPU
- ② parallelize
 - `teams` and `distribute`
 - `teams` : creates a number of teams executing the same statement (\approx `parallel` pragma)
 - `distributed` : distribute iterations of a for loop among teams (\approx `for` pragma)
 - `parallel` and `for`
 - `parallel` : creates a number of threads executing the same statement in a team
 - `for` : distribute iterations of a for loop among threads in a team
 - think of `teams` + `distributed` another layer outside `parallel` + `for`
- ③ move (or sync) data
 - `target data` : move/sync data between CPU and GPU

Implementation note

- while not specified anywhere in the spec (and there are cases they behave differently to below), you can think of
 - a team \sim a thread block
 - a thread \sim a CUDA thread
- it at least helps you understand why things look so redundant ...

Frequently-used combined idioms

- all combined

```
1 #pragma omp target teams distribute parallel for
2 for (int i = start; i < end; i += incr) {
3     S
4 }
```

- teams + distributed to outer loop and parallel + for to inner loop

```
1 #pragma omp target teams distribute
2 for (int i = start; i < end; i += incr) {
3     #pragma omp parallel for
4     for (int j = start'; j < end'; j += incr') {
5         S
6     }
7 }
```

- similar to launching a kernel doing S , but
 - you don't have to adjust thread block size
 - the program is orthogonal to thread count

Data mapping

a major headache when programming in CUDA is data management

- the only “transparent” data transfer is argument passing

```
1 f<<<nb,bs>>>(a, b, c, ...);
```

- getting the result back from GPU is already painful

```
1 cudaMalloc(&r_dev, ...);  
2 f<<<nb,bs>>>(a, b, c, ..., r);  
3 cudaMemcpy(r, r_dev, ...);
```

- for persistent data,
 - maintain two pointers to logically same data (CPU version and GPU version)
 - get them synched when necessary (before and after a kernel launch)

“data mapping” of OpenMP alleviates the pain

Data mapping example

```
1 #pragma target data map(to: a[b:c]) map(from: x)  
2 S
```

- send the array range $a[b:c]$ ($a[b]$, $a[b+1]$, ..., $a[c-1]$) to GPU before S
- send x from GPU after S
- you can combine **to:** and **from:** into **tofrom:**
- somewhat “declarative” way of understanding this
 - expressions $a[i]$ ($b \leq i < c$) become valid (“mapped”) on GPU during S
 - expressions x become valid on CPU after S
- note: you can specify **map** clauses as part of **target** (not **target data**) directive, too
- learn details with tht notebook