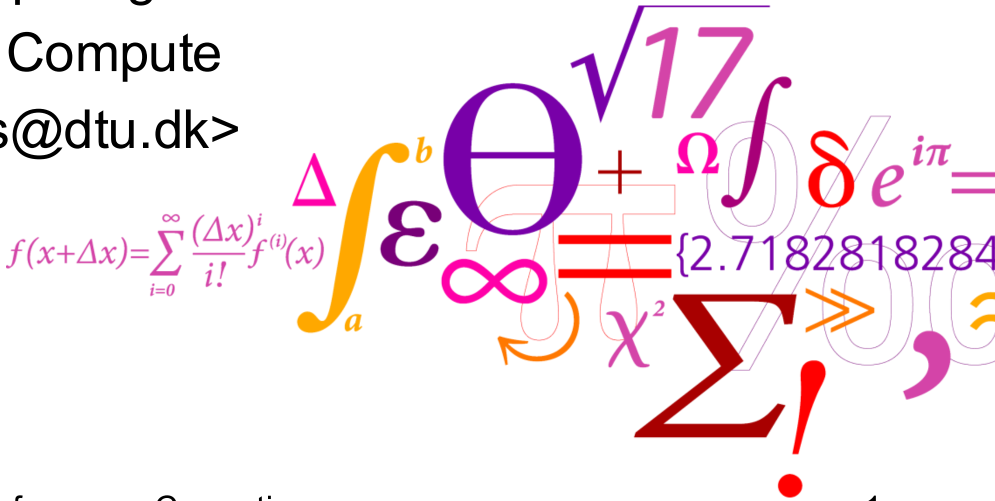
GPU Programming with OpenMP Part1: OpenMP offload basics

Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>



Overview

- GPU programming languages
- GPU programming model
- OpenMP offload basics
  - Target constructs
  - Work-sharing constructs
  - Synchronization
- Runtime library API / Environment variables

Many ways to program GPUs

- CUDA (2007)
- OpenCL (2009)
- OpenACC (2012)
- OpenMP 4.0 (2013)
- Others (ROCm, Numba, oneAPI, Matlab,..)

What is CUDA?

- [**C**ompute **U**nified **D**evice **A**rchitecture]
- A parallel computing standard and API proposed by NVIDIA for general-purpose computations on CUDA-enabled GPUs
  - Priority #1: Make things easy (Sell GPUs)
  - Priority #2: Get performance
- Result: Low level – but mainly C++ syntax
  - Requires expert knowledge to get best performance
- Scalable
- Well documented and free to use (!)

What is OpenCL?

- **Open C**omputing **L**anguage (current v3.0)
- Khronos group (non-profit organization):
  - “*OpenCL is an open, royalty-free standard for cross-platform, parallel programming of modern processors found in parallel computers, servers and handheld/embedded devices.*”
- Open standard for heterogeneous computing
- Priority #1: Become the *industry-wide future standard* for heterogeneous computing
- Priority #2: Use all computational resources in the system efficiently
- Up to vendors to provide support!

CUDA vs OpenCL?

- Most CUDA features map one-to-one to OpenCL features (only the syntax is different)
- CUDA comes with a mature software framework
- CUDA comes with tuned high-performance libs
    - cuBLAS – CUDA Basic Linear Algebra Subroutines library
    - cuFFT – CUDA Fast Fourier Transform library
    - cuSPARSE – CUDA Sparse Matrix library .... and many more!
  - OpenCL have had much less effort in this direction
- CUDA is well documented (by NVIDIA)
- NVIDIA products are widely used in HPC (>90%)
  - OpenCL still lags in performance for NVIDIA products

What is OpenACC?

- OpenACC is an open specification for compiler directives for parallel programming

  `#pragma acc directive [clause]`

- Developed by PGI, Cray, CAPS, and NVIDIA
- High-level directives: Minimal modifications to the code, fever lines than with CUDA, OpenCL,..
- Supports CPUs, GPU accelerators and co-processors from multiple vendors
- Compiler support: `gcc`, `nvc++` (`pgi`), `cc`, `clacc`

OpenMP for GPUs

- OpenMP is an open specification for compiler directives for parallel programming

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

- OpenMP 4.0 – 6.0: Standardizes established practice for heterogeneous device programming
- Support in all common compilers
- Behind the scenes: Intermediate level of CUDA or OpenCL is typically used for GPUs

OpenMP vs OpenACC?

- OpenMP is the established 'de-facto' standard
  - You know the directives and terminology from week 2
  - Let's not switch and complicate things…
- PGI OpenACC was known for good performance
  - Few years ago: OpenACC + `pgi` compiler had generally better performance than OpenMP + `gcc` / `clang`!
  - Now: OpenMP + `nvc++` gives similar high performance
- Why not CUDA?
  - Low level (!) – week 3 tends to be debug, debug..
  - If you want to learn CUDA please sign up for the CUDA special course that will run in the spring semester 2026

NVIDIA HPC compiler

- The NVIDIA HPC C++ compiler is called
  `nvc++ [options] [path]filename [...]`
- Many options are common with gcc: `-g -fast` ..
- Compiles for OpenMP, OpenACC, and CUDA
  - `-mp=gpu`
  - `-acc` (can be used together)
  - `-cuda`
- If you run into linking problems (undefined refs.)
  - Avoid mixing `.c` and `.cpp` files (`.cpp` and `.cu` is fine)
  - Or use `extern ″C″ { … }` apropriately

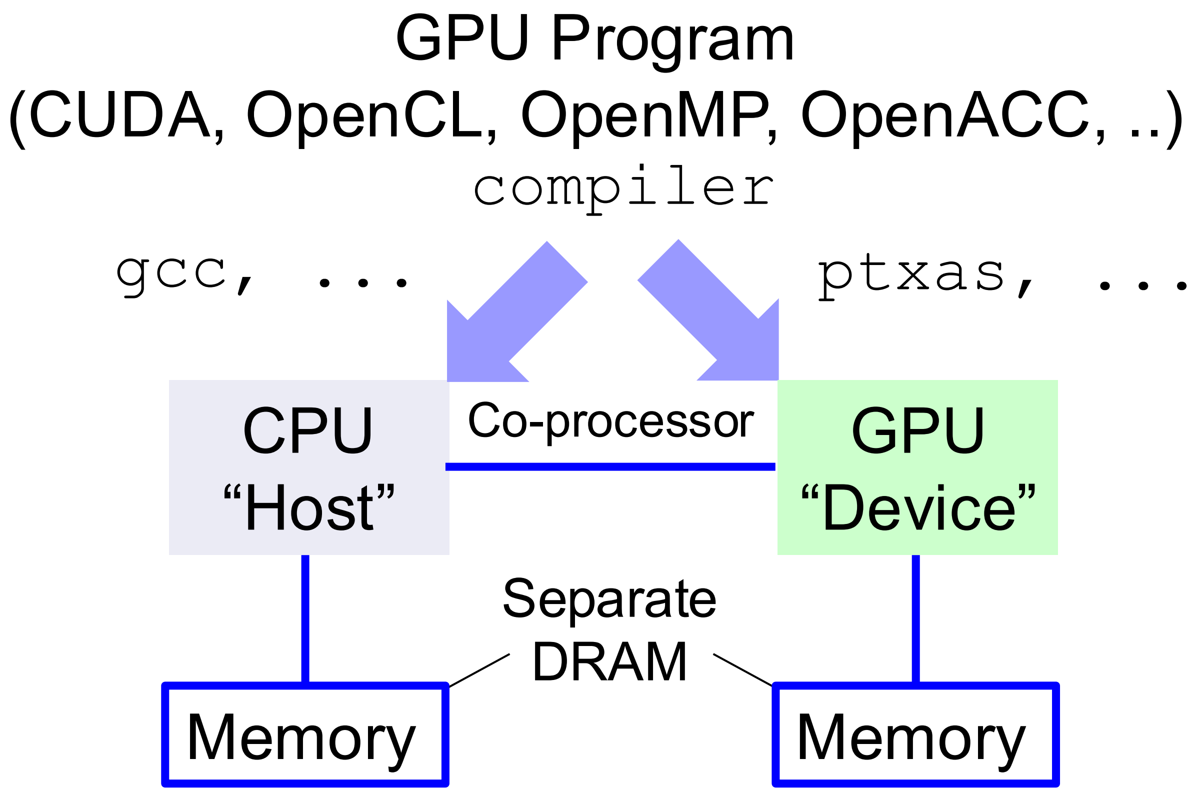
NVIDIA HPC compiler

- We provide a `Makefile` template for exercises!
- Most important compiler flags:
- `-gpu=cc90`
  - Compile code for compute capability 9.0 (Hopper)
  - Default is cc. 1.0 (Tesla), latest is cc. 9.x (Hopper)
- `-Minfo`
  - Set output comments from compiler to verbose
- `-gpu=lineinfo`
  - Generate line-number information for device code (e.g., used in Nsight™ profiler)

https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide

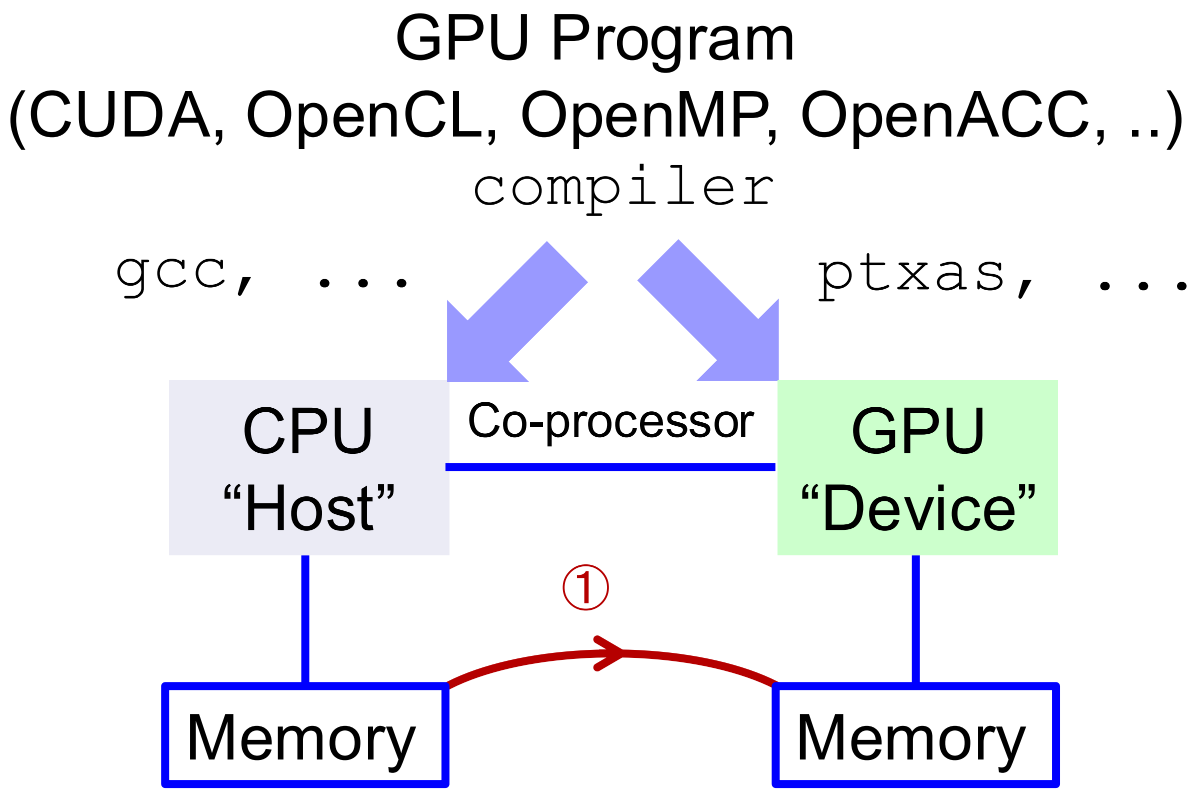
GPU programming model

GPU programming model



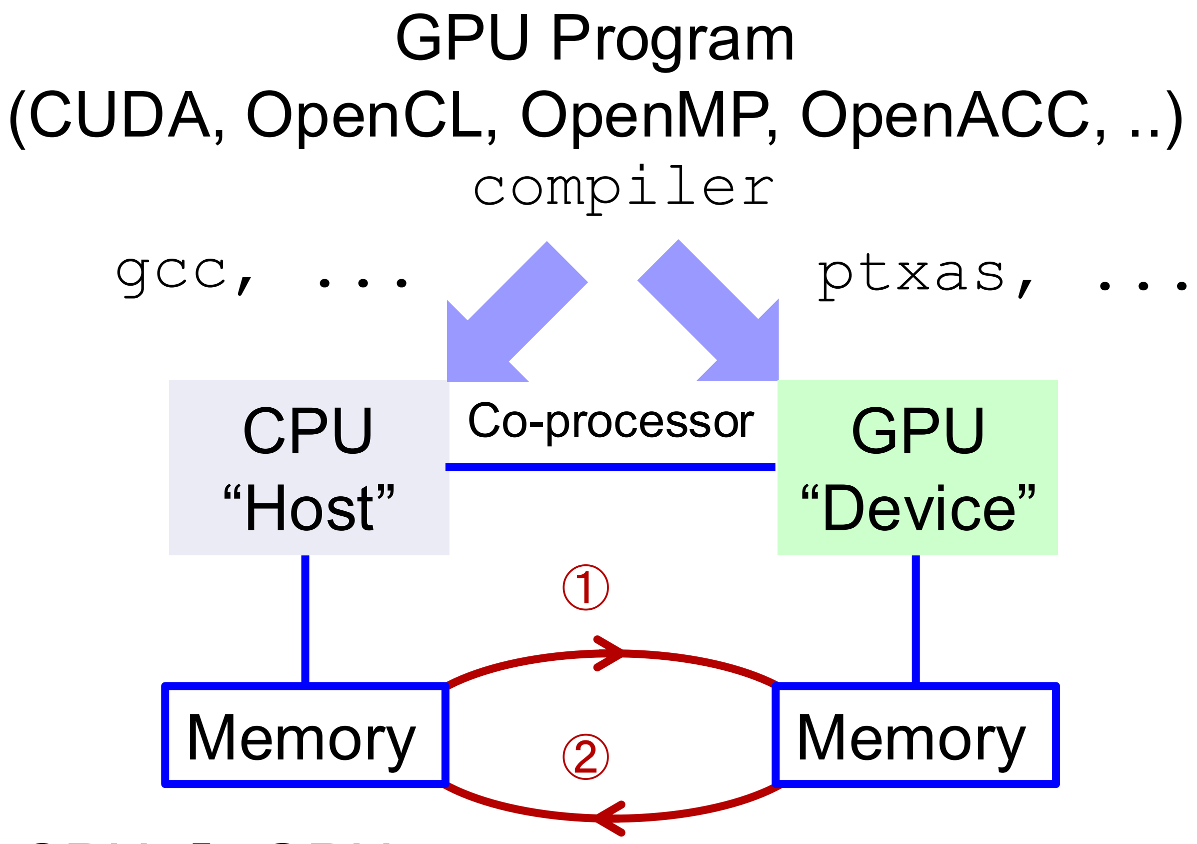
- Host – the CPU
  - In charge, manages resources
  - Runs main(), etc.
- Device – the GPU
  - Co-processor / accelerator
  - Runs specific tasks

GPU programming model



① Data CPU ➔ GPU

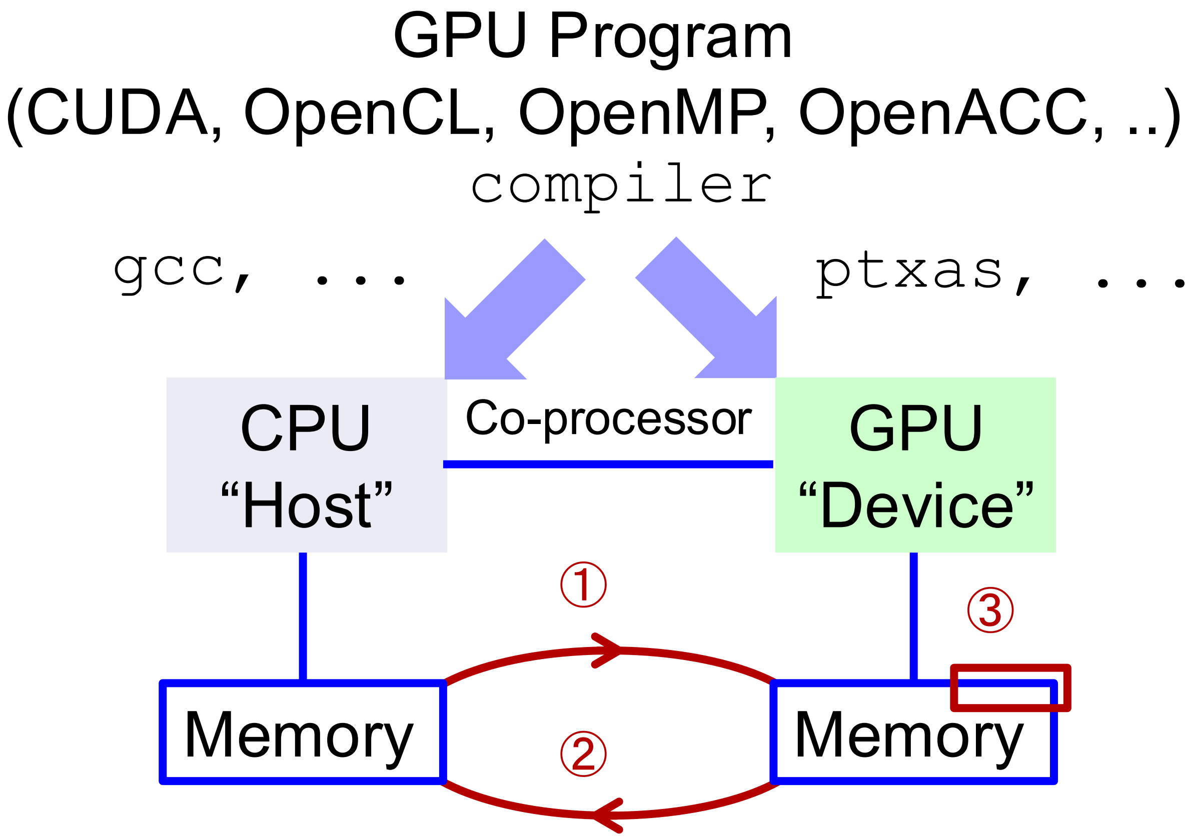
GPU programming model



① Data CPU ➔ GPU

② Data GPU ➔ CPU

GPU programming model

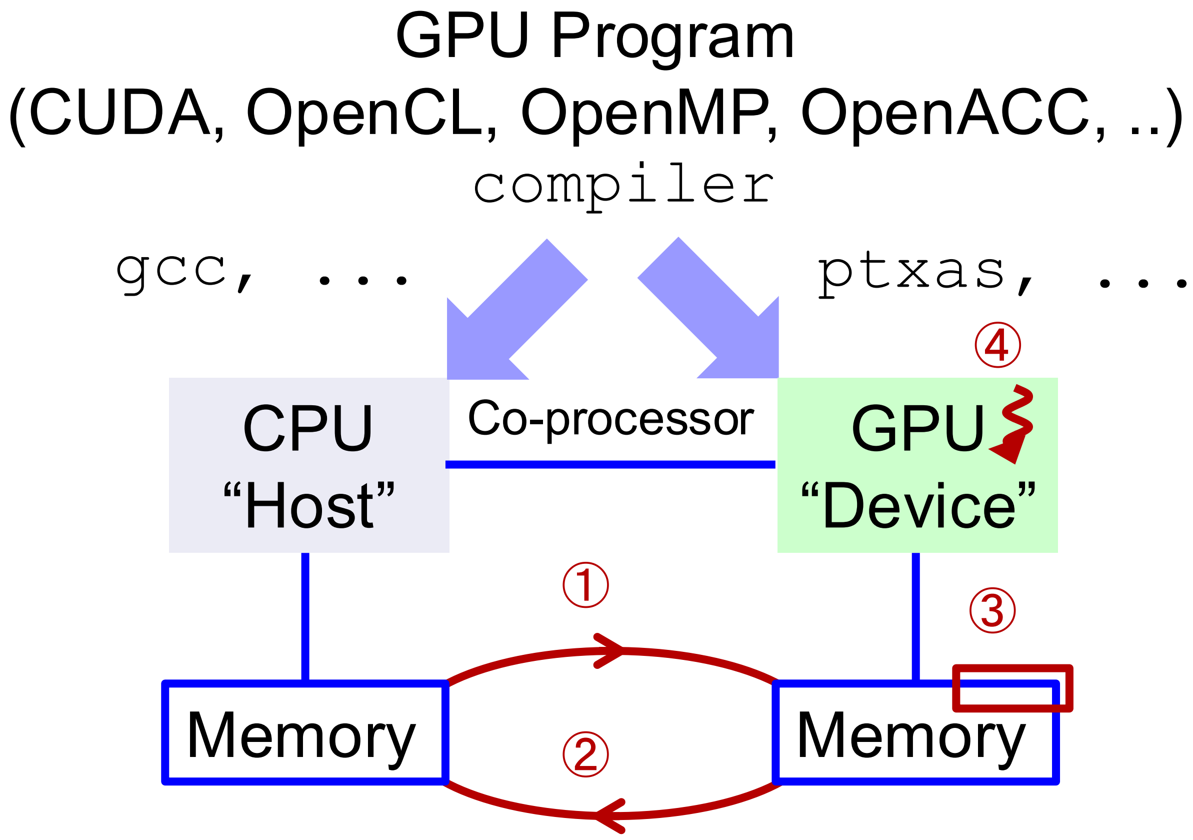


① Data CPU ➔ GPU

② Data GPU ➔ CPU

③ Allocate GPU memory

GPU programming model



① Data CPU → GPU

② Data GPU → CPU

③ Allocate GPU memory

④ Offload code to device

OpenMP offload basics

OpenMP offload basics

- Syntax C/C++:

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

- Clause can be
  - `private(list)`
  - `firstprivate(list)`
  - `if([target:] scalar_expr)`
  - `nowait`
  - `depend([modifier,] type: list)`
  - `allocate([allocator:] list)`
  - `in_reduction(op: list)`
  - `thread_limit(int_expr)`
  - `device([modifier:] int_expr)`
  - `map([{alloc | to | from | tofrom}:] list)`
  - `defaultmap(behavior[:var-category])`
  - `is_device_ptr(list)`
  - `has_device_addr(list)`

OpenMP offload basics

- Syntax C/C++:

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

Not currently supported in `nvc++`

- Clause can be
  - `private(list)`
  - `firstprivate(list)`
  - `if([target:] scalar_expr)`
  - `nowait`
  - `depend([modifier,] type: list)`
  - ~~`allocate([allocator:] list)`~~
  - ~~`in_reduction(op: list)`~~
  - `thread_limit(int_expr)`
  - `device([modifier:] int_expr)`
  - `map([{alloc | to | from | tofrom}:] list)`
  - `defaultmap(behavior[:var-category])`
  - `is_device_ptr(list)`
  - `has_device_addr(list)`

OpenMP offload basics

- First OpenMP offload “Hello world”:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp target
    {
        printf("Hello world from the device!\n");
    } // end target
    return(0);
}
```

OpenMP offload basics

```
$ nvc++ -mp=gpu -Minfo -o hello hello.cpp
main:
     11, #omp target
         11, Generating "nvkernel_main_F1L11_2" GPU
kernel

$ ./hello
Hello world from the device!
```

- Note: The compiler generates a (CUDA) kernel to run on the GPU with a specific name starting with `nvkernel_[function name]_`
- The programmer need to explicitly create parallelism on the device – `target` is not enough

OpenMP offload basics

- Syntax C/C++:

```
#pragma omp target teams [clause]
{
      ...
}
```

- Extra clauses with `teams` are
  - `shared(list)`
  - `default(behavior)`
  - `reduction(op: list)`
  - `num_teams(int_expr)`

(implementation defined limit = 65536 teams)

OpenMP offload basics

- Second OpenMP offload “Hello world”:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp target teams
    {
        printf("Hello world from the device!\n");
    } // end target teams
    return(0);
}
```

OpenMP offload basics

```
$ ./hello
Hello world from the device!
Hello world from the device!
...
Hello world from the device!

$ ./hello | wc -l
114
```

- By default construct `target teams` creates as many teams as there are compute units (SMs) on the device
- The teams are scheduled to run independently
- Only a single thread in each team executes the code

OpenMP offload basics

- Syntax C/C++:

```
#pragma omp target parallel [clause]
{
     ...
}
```

- Extra clauses with `parallel` are
  - `proc_bind(master | close | spread)`

OpenMP offload basics

- Third OpenMP offload “Hello world”:

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

int main(int argc, char *argv[]) {
    #pragma omp target parallel
    {
        printf("Hello world from %d!\n”,
                omp_get_thread_num());
    } // end target parallel
    return(0);
}
```

OpenMP offload basics

```
$ ./hello
Hello world from 4960!
Hello world from 4961!
...
Hello world from 10944!

$ ./hello | wc -l
14592
```

- Note: The order of execution will be different from run to run (but in “groups” of 32)!
- The default no. of threads depends on the OpenMP implementation and the hardware
- Here it starts $114 * 4 * \underbrace{32}_{}$ = 14592 threads in 1 team

[CUDA] number of threads in a warp!

OpenMP offload basics

- Syntax C/C++:

```
#pragma omp target teams parallel [clause]
{
    ...
}
```

- No extra clauses

OpenMP offload basics

- Final OpenMP offload “Hello world”:

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    #pragma omp target teams parallel \
        num_teams(114) thread_limit(4*32)
    {
         printf("Hello world from (%d, %d)!\n",
                omp_get_team_num(),
                omp_get_thread_num());
    } // end target teams parallel
    return(0);
}
```

OpenMP offload basics

```
$ nvc++ -mp=gpu -Minfo -o hello hello.cpp
main:
     22, #omp target teams num_teams(114) thread_limit(128)
         22, Generating "nvkernel_main_F1L22_2" GPU kernel
             #omp parallel

$ ./hello
Hello world from (4, 64)!
Hello world from (4, 65)!...
Hello world from (74, 31)!

$ ./hello | wc -l
14592
```

- If we did not specify anything it would start 114 teams of 992 threads = 114 * 31 * 32 = 113088 threads

[CUDA] just below the maximum number of blocks per SM!

OpenMP offload basics

Work-sharing constructs – `loop`:

- Syntax C/C++:

```
#pragma omp target teams loop [clause]
for-loop
```

- `loop` asserts the ability of a loop to be run in any order, finding the available parallelism
- Clause can be any of the clauses accepted by the `target`, `teams` or `loop` directives with identical meanings and restrictions

OpenMP offload basics

Work-sharing constructs – `loop`:

- `omp target teams loop`
  - Recommended way
  - Can use `num_teams` and `thread_limit` clauses
- `omp target loop`
  - Fully automatic
  - No `num_teams` and `thread_limit` clauses
- `omp target parallel loop`
  - Uses only threads, and doesn't use teams
  - Might be useful for light computation

OpenMP offload basics

Work-sharing constructs – `distribute`:

- Syntax C/C++:

```
#pragma omp target teams distribute \
  parallel for [clause]
for-loop
```

- The iterations are distributed and executed in parallel by all threads of the teams
- Clause can be any of the clauses accepted by the `target`, `teams`, `distribute` or `parallel for` directives with identical meanings and restrictions

OpenMP offload basics

Work-sharing constructs – `distribute`:

```
#define N 16
int main(int argc, char *argv[]) {
    double a[N], b[N], c[N];
    for (int i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    #pragma omp target teams \
        distribute parallel for
     for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];

    for (int i = 0; i < N; i++)
        printf("%f\n", c[i]);
}
```

- Will this run?
  - Yes! – OpenMP 6.0: “implicit data-mapping rules”
- Is it efficient?
  - No!

OpenMP offload basics

- Synchronization – remember for last week:
  - `#pragma omp single`
  - `#pragma omp critical`
  - `#pragma omp atomic`
  - `#pragma omp ordered`
  - `#pragma omp barrier` ←
- Implied barriers ← also current team
  - exit from parallel region
  - exit from omp for/omp do/omp workshare
  - exit from sections
  - exit from single

OpenMP 6.0 specification: "The binding thread set for a barrier region is the current **team**."

OpenMP offload basics

- Synchronization – remember for last week:
  - `#pragma omp single`
  - ~~`#pragma omp critical`~~
  - `#pragma omp atomic`
  - ~~`#pragma omp ordered`~~
  - `#pragma omp barrier`

Not currently supported in `nvc++`

OpenMP 6.0 specification: "The binding thread set for a barrier region is the current **team**."

also current team

- Implied barriers
  - exit from parallel region
  - exit from omp for/omp do/omp workshare
  - ~~exit from sections~~
  - exit from single

OpenMP offload basics

- What does this mean in practice

working host version

```
int count = 0;
#pragma omp parallel \
    num_threads(16)   \
    shared(count)
{
    #pragma omp atomic
    count += 1;

    // Wait for all threads done
    #pragma omp barrier

    #pragma omp master
    printf("# of threads is %d\n",
           count);
}
```

incorrect device version

```
int count = 0;
#pragma omp target teams parallel \
   num_teams(114) thread_limit(64)\
   map(tofrom:count)
{
    #pragma omp atomic
    count += 1;

    // Wait for all threads done
    #pragma omp barrier

    #pragma omp master
    printf("# of threads is %d\n",
           count);
}
```

OpenMP offload basics

- What does this mean in practice

```
$ ./threads
# of threads is 16

$ ./threads
# of threads is 16

$ ./threads
# of threads is 16
```

```
$ ./threads_offload
# of threads is 5925
# of threads is 5945
[... 111 ...]
# of threads is 6319

$ ./threads_offload
# of threads is 5886
# of threads is 5904
[... 111 ...]
# of threads is 6319
```

OpenMP offload basics

- What does this mean in practice

working device version

```
int count = 0;
#pragma omp target teams parallel \
   num_teams(114) thread_limit(64)\
   map(tofrom:count)
{
    #pragma omp atomic
    count += 1;
}
// To wait for all threads done we
   use two separate offload regions
#pragma omp target map(to:count)
{
    printf("# of threads is %d\n",
           count);
}
```

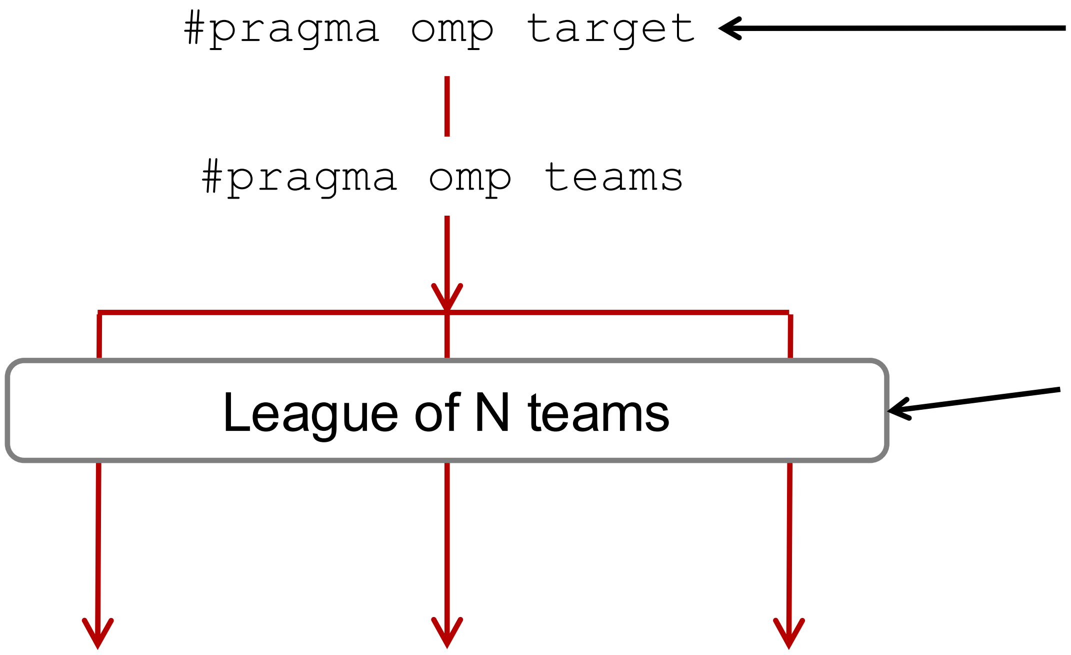
```
$ ./threads_offload
# of threads is 7296

$ ./threads_offload
# of threads is 7296
```

- Synchronization among all threads requires two separate offload regions

OpenMP offload basics summary

`#pragma omp target` ← Execute code on device

OpenMP offload basics summary

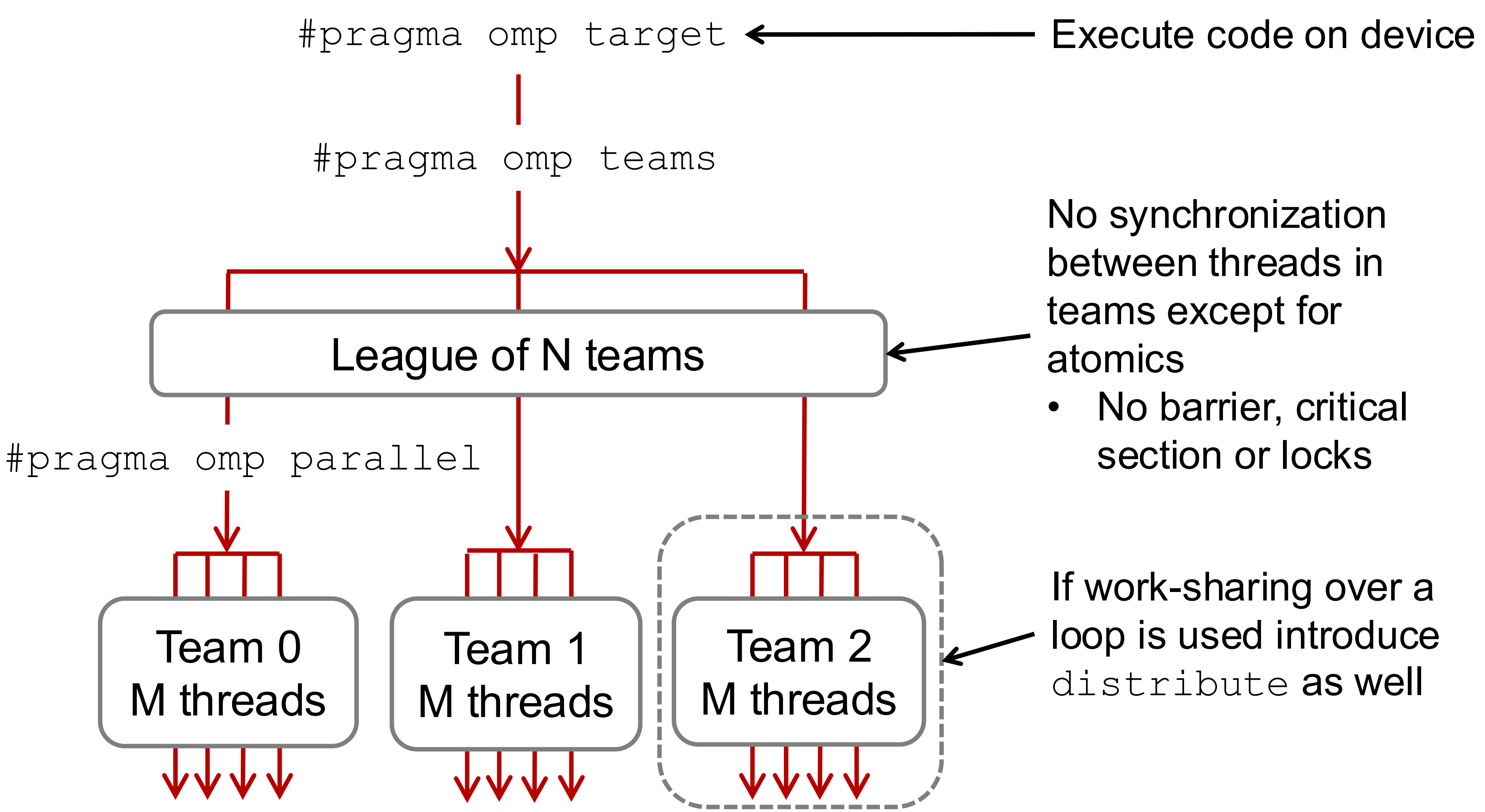


Execute code on device

No synchronization between threads in teams except for atomics

- No barrier, critical section or locks

OpenMP offload basics summary



No synchronization between threads in teams except for atomics

- No barrier, critical section or locks

If work-sharing over a loop is used introduce `distribute` as well

OpenMP runtime library

OpenMP offload runtime library

- New library routines

Name	*Functionality*
`int omp_get_num_teams(void)`	get number of teams
`int omp_get_team_num(void)`	get team number
`int omp_get_num_devices(void)`	get number of devices
`void omp_set_default_device(int dev_num)`	set the default device
`int omp_get_default_device(void)`	get the default device
`int omp_get_initial_device(void)`	get initial device (=host)
`int omp_is_initial_device(void)`	are we on the host?
`int omp_get_num_procs(void)`	get number of processors

for more details see the OpenMP 5.0 specifications (https://www.openmp.org/specifications/)

OpenMP offload runtime library

- New library routines

Name	*Functionality*
`void* omp_target_alloc(size_t, int dev_num)`	allocate memory on device
`void omp_target_free(void*, int dev_num)`	free memory on device
`int omp_target_memcpy(...., int dev_num)`	memcpy to and from device
`int omp_target_memcpy_rect(...., int dev_num)`	memcpy to and from device of a rectangular subvolume
`omp_target_associate_ptr(...)` `omp_target_disassociate_ptr(...)` `omp_target_is_present(...)`	combining device ptr with host ptr to be used in map clause (we use is_device_ptr clause)

Offload environment variables

- `OMP_DEFAULT_DEVICE = n`
  - Sets the default device when “device(n)” clause is not specified (error if n >= no. of devices)
- `OMP_NUM_TEAMS = n`
  - Maximum number of `num_teams` allowed
- `NVCOMPILER_OMP_CUDA_GRID=teams,limit`
  - Force a specific `num_teams` and `thread_limit`
- `OMP_TARGET_OFFLOAD = [mandatory | disabled | default]`
  - Controls whether offload region runs on device or host

Warmup of GPUs

- It takes time to get a context on a device
  - Idle GPUs are in power saving mode
  - Just-in-time compilation (CUDA)
  - Transfer of kernel to GPU memory
  - Approx. 0.2 seconds (on our nodes)
- Warmup run
  - Required for accurate performance benchmarking in case of short runtimes
  - First offload that modifies the GPU context will initiate ‘warm up’ of the device
  - First transfer of data starts the device data environment

Exercises

- Do the first exercise
  - ex1_nvaccelinfo
  - Please note that `nvc++` v25.11 requires `gcc` version 15.x or older and CUDA 13.0 or older (so please do not load the newer versions even though they exist)
- Then start the second exercise
  - ex2_helloworld
  - Template `Makefile` available on DTU Learn
- Next lecture at 13.00 (Monday)!

Acknowledgements

- Some slides are from Jeff Larkin, NVIDIA's HPC Software team:
  - https://developer.nvidia.com/blog/author/jlarkin/
- Some slides are from Michael Klemm, OpenMP Architecture Review Board, AMD:
  - https://www.openmp.org/about/our-team/officers-and-staff/
- Some slides are from “OpenMP 4.5 target” by Tom Scogland and Oscar Hernandez:
  - ECP OpenMP tutorial 06-28-2017
- Some slides are from NVIDIA Developer
  - https://developer.nvidia.com/

End of lecture