Introduction to OpenMP - GPUs

International High-Performance Computing Summer School 2023

Ludovic Capelli

EPCC

July 11, 2023

epcc

Individuals

The material presented in this lecture is inspired from content developed by:

- John Urbanic
- Michael Klemm

Contributors

In this material:

- The slides are created using LaTeX *Beamer* available at `https://ctan.org/pkg/beamer`.
- The sequence diagrams are created using an extended version of the *pgf-umlsd* package available at `https://ctan.org/pkg/pgf-umlsd`.

License - Creative Commons BY-NC-SA-4.0[1]

Your opinion matters

Figure: Link to surveys

Timetable

First session: Monday

- Topic: CPUs
- From 14:00 until 15:30
- 30-min break
- From 16:00 until 17:30

Timetable

First session: Monday

- Topic: CPUs
- From 14:00 until 15:30
- 30-min break
- From 16:00 until 17:30

Second session: Tuesday

- Topic: GPUs
- From 9:00 until 10:30
- 30-min break
- From 11:00 until 12:30

Moment of truth

Before we start:

- Connect to bridges

```
ssh your_username@bridges2.psc.edu
```

- Clone the repository used in this session

```
git clone https://github.com/capellil/
    IHPCSS_Introduction_to_OpenMP_GPU_examples
```

Moment of truth

- You will see it contains multiple folders, numbered.
- Each of which will be an illustration to a concept we will see.

Moment of truth

- Load the modules needed for the GPU ecosystem.

```
module load nvhpc/22.9;
module load openmpi/4.0.5-nvhpc22.9;
```

- Put these two lines in your `.bashrc` file.

Moment of truth

- Go to the repository, inside the first folder.

```
cd repository/<language>/1.Preamble
```

- Compile the source code in it

```
make
```

- Run the executable on the compute node

```
sbatch submit.slurm
```

- To see whether your code is executing, has finished, etc...

```
watch squeue -u $USER
```

Table of content

Table of Contents

What are GPUs?

What are GPUs?

- Stands for **G**raphics **P**rocessing **U**nits.

What are GPUs?

- Stands for **G**raphics **P**rocessing **U**nits.
- Devices specialised in... graphics processing.

What are GPUs?

- Stands for **G**raphics **P**rocessing **U**nits.
- Devices specialised in... graphics processing.
- Typically has a given operation to apply to a lot of inputs
  - points in a mesh
  - pixels in a picture

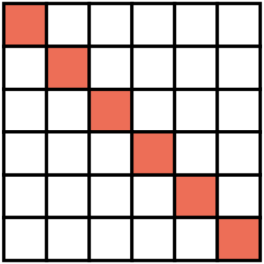
What are GPUs?

- Stands for **G**raphics **P**rocessing **U**nits.
- Devices specialised in... graphics processing.
- Typically has a given operation to apply to a lot of inputs
  - points in a mesh
  - pixels in a picture
- The GPU architecture allows them to complete this task very effectively.

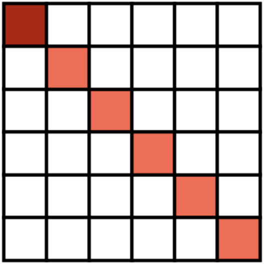
Use case

- Imagine you are developing a graphic editing software.
- You are currently implementing a filter allowing to darken a picture by making every pixel 20% darker.
- The same calculation is applied to every pixel, regardless of its current colour.

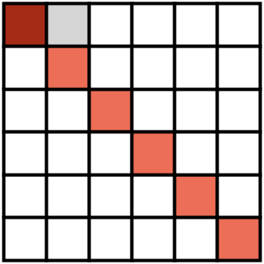
How CPUs would do it



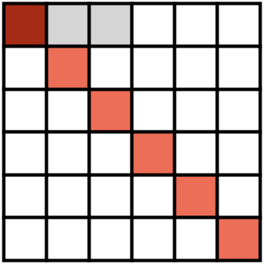
How CPUs would do it.



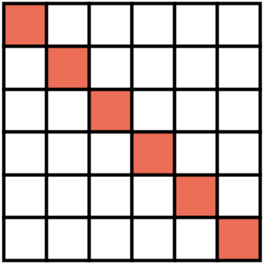
How CPUs would do it



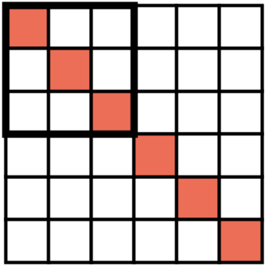
How CPUs would do it



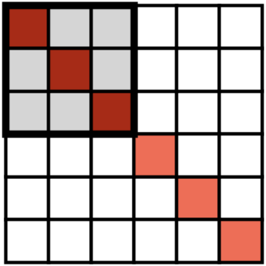
How GPUs work.



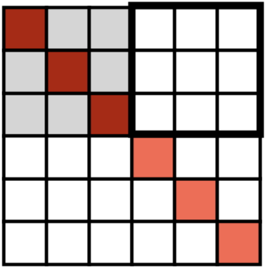
How GPUs work.



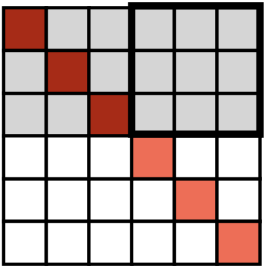
How GPUs work.



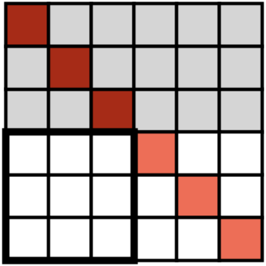
How GPUs work.



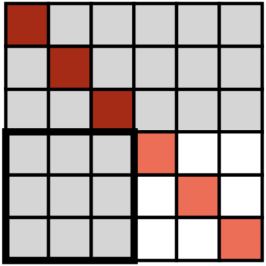
How GPUs work.



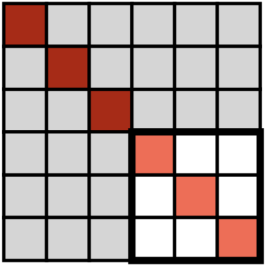
How GPUs work.



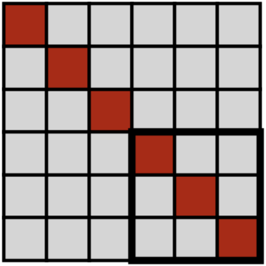
How GPUs work.



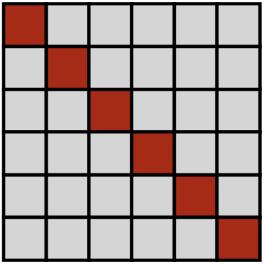
How GPUs work.



How GPUs work.



How GPUs work.



How GPUs work.

Demonstration by Mythbusters and NVIDIA at https://www.youtube.com/watch?v=-P28LKWTzrI.

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.
  - Core clock: 1.3GHz, boost at 1.5GHz.

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.
  - Core clock: 1.3GHz, boost at 1.5GHz.
  - 5,120 cores.

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.
  - Core clock: 1.3GHz, boost at 1.5GHz.
  - 5,120 cores.
  - Bandwidth: 900GB/s

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.
  - Core clock: 1.3GHz, boost at 1.5GHz.
  - 5,120 cores.
  - Bandwidth: 900GB/s
  - Max Thermal Design Power: 250W.

GPU specs

- On Bridges2, the GPU nodes have NVIDIA V100-32GB SXM2.
  - Core clock: 1.3GHz, boost at 1.5GHz.
  - 5,120 cores.
  - Bandwidth: 900GB/s
  - Max Thermal Design Power: 250W.
- Each GPU node on Bridges2 has 8 such GPUs...
  - Total of over 40K+ GPU cores per node
  - On top of the two Intel Xeon Gold 6248 "Cascade Lake" CPUs per node

Advantages of GPUs

Advantages of GPUs

- Computation tanks: 15.7 TFLOPS per V100GPU, over 120 TFLOPS per GPU node.

Advantages of GPUs

- Computation tanks: 15.7 TFLOPS per V100GPU, over 120 TFLOPS per GPU node.
- Higher power efficiency: 250W per GPU.

Advantages of GPUs

- Computation tanks: 15.7 TFLOPS per V100GPU, over 120 TFLOPS per GPU node.
- Higher power efficiency: 250W per GPU.
- Low price per core:
  - £5,400 for 1 CPU that has 20 cores, so ∼£270 per core.

Advantages of GPUs

- Computation tanks: 15.7 TFLOPS per V100GPU, over 120 TFLOPS per GPU node.
- Higher power efficiency: 250W per GPU.
- Low price per core:
  - £5,400 for 1 CPU that has 20 cores, so ∼£270 per core.
  - £4,000 for 1 GPU that has 5,000 cores, so ∼£1 per core.

Advantages of GPUs

- Computation tanks: 15.7 TFLOPS per V100GPU, over 120 TFLOPS per GPU node.
- Higher power efficiency: 250W per GPU.
- Low price per core:
  - £5,400 for 1 CPU that has 20 cores, so ∼£270 per core.
  - £4,000 for 1 GPU that has 5,000 cores, so ∼£1 per core.

Why even bothering with CPUs then?

Limitations

[2] though alleviated now with unified memory, which in turns raises new challenges as now the CPUs and the GPUs can have race conditions between them

Limitations

- Memory transfers[2]

[2] though alleviated now with unified memory, which in turns raises new challenges as now the CPUs and the GPUs can have race conditions between them

Limitations

- Memory transfers[2]
- Do not handle divergence well (needs to get cores idle)

[2] though alleviated now with unified memory, which in turns raises new challenges as now the CPUs and the GPUs can have race conditions between them

Limitations

- Memory transfers[2]
- Do not handle divergence well (needs to get cores idle)
- Simpler **I**nstruction **S**et **A**rchitecture (ISA): operations issued do not support as advanced calculations

[2] though alleviated now with unified memory, which in turns raises new challenges as now the CPUs and the GPUs can have race conditions between them

Limitations

- Memory transfers[2]
- Do not handle divergence well (needs to get cores idle)
- Simpler **I**nstruction **S**et **A**rchitecture (ISA): operations issued do not support as advanced calculations
- Non-coalesced memory accesses

[2] though alleviated now with unified memory, which in turns raises new challenges as now the CPUs and the GPUs can have race conditions between them

Opportunity to seize

- Despite having limitations, GPUs still offer a highly attractive solution to certain types of workloads, especially in scientific simulations.

Opportunity to seize

- Despite having limitations, GPUs still offer a highly attractive solution to certain types of workloads, especially in scientific simulations.
- Decided to harness that graphical pipeline for more general computing tasks.

Opportunity to seize

- Despite having limitations, GPUs still offer a highly attractive solution to certain types of workloads, especially in scientific simulations.
- Decided to harness that graphical pipeline for more general computing tasks.
- Birth of GPGPU: General Purpose Graphical Processing Unit.

Solutions

CUDA First on it, free, amazing documentation, but only available for NVIDIA GPUs. Imposed their terms: CUDA cores, CUDA threads, warps etc...

Solutions

CUDA
: First on it, free, amazing documentation, but only available for NVIDIA GPUs. Imposed their terms: CUDA cores, CUDA threads, warps etc...

OpenACC
: **Open Acc**elerators. Directive-based solution for GPGPU. Limited support in compilers.

Solutions

CUDA First on it, free, amazing documentation, but only available for NVIDIA GPUs. Imposed their terms: CUDA cores, CUDA threads, warps etc...

OpenACC **Open Acc**elerators. Directive-based solution for GPGPU. Limited support in compilers.

OpenMP NVIDIA support on their GPUs noticeably improved recently with OpenMP 5.0, this is why we will use it, now that you already know how to use it for CPUs.

Ideas

- `declare target` construct: associated variables and function are to be mapped onto the target devices (thus usable in device code).
- passing the `device_type(nohost)` clause to a `declare target` construct forces the compiler to not produce host code for the associated structured block (pertaining to both variables and functions included)

Table of Contents

Host vs device[3]

Host The device on which the OpenMP program begins execution.

[3] See Section 1.2.1 in OpenMP standard version 5.2

Host vs device[3]

Host The device on which the OpenMP program begins execution.

Target A device onto which code and data may be offloaded from the host device.

[3] See Section 1.2.1 in OpenMP standard version 5.2

`target` construct

- **Transfers** the control flow to the target device

`target` construct

- **Transfers** the control flow to the target device
- This transfer is:

`target` construct

- **Transfers** the control flow to the target device
- This transfer is:
  - sequential

`target` construct

- **Transfers** the control flow to the target device
- This transfer is:
  - sequential
  - synchronous

`target` construct

- **Transfers** the control flow to the target device
- This transfer is:
  - sequential
  - synchronous
- **In theory**, this can be combined with any OpenMP construct.

`target` construct

- **Transfers** the control flow to the target device
- This transfer is:
  - sequential
  - synchronous
- **In theory**, this can be combined with any OpenMP construct.
- **In practice**, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Offloading to a device - Example

```c
// Block A: executed on host
// Block B: executed on host
// Block C: executed on host
```

```fortran
! Block A: executed on host
! Block B: executed on host
! Block C: executed on host
```

Offloading to a device - Example

```c
// Block A: executed on host
#pragma omp target
{
    // Block B: executed on target
}
// Block C: executed on host
```

```fortran
! Block A: executed on host
!$OMP TARGET
    ! Block B: executed on target
!$OMP END TARGET
! Block C: executed on host
```

Currently on the host or device?

- When you will start to have functions, and nested functions, you may not know at some point whether you are on the host or on a device.
- You can check by calling the function `omp_is_initial_device()`.

Currently on the host or device?

```c
if(omp_is_initial_device())
{
    printf("I am on the host\n");
}
else
{
    printf("I am on the device.\n");
}
```

```fortran
IF(omp_is_initial_device())
    WRITE(*, '(A)') 'I am on the host.'
ELSE
    WRITE(*, '(A)') 'I am on the device.'
END IF
```

Time to practise: `2.HelloWorld`

Update the source code provided such that the for loop marked is executed on the GPU.

Tips

You will need:

- the `target` construct

Table of Contents

Implicit mapping - CPUs

```c
int a = 123;
int b[2] = {456, 789};
#pragma omp parallel
{
    a *= 2;
    b[0]++;
    b[1]++;
}
```

```fortran
INTEGER :: a = 123
INTEGER, DIMENSION(0:1) :: b := (/456, 789/)
!$OMP PARALLEL
    a *= 2
    b(0) = b(0) + 1
    b(1) = b(1) + 1
!$OMP END PARALLEL
```

Implicit mapping - CPUs

```c
int a = 123;
int b[2] = {456, 789};
#pragma omp parallel default(none) shared(a, b)
{
    a *= 2;
    b[0]++;
    b[1]++;
}
```

```fortran
INTEGER :: a = 123
INTEGER, DIMENSION(0:1) :: b = (/456, 789/)
!$OMP PARALLEL DEFAULT(NONE) SHARED(a, b)
    a *= 2;
    b(0) = b(0) + 1
    b(1) = b(1) + 1
!$OMP END PARALLEL
```

Implicit mapping - GPUs

```c
int a = 123;
int b[2] = {456, 789};
#pragma omp target
{
    a *= 2;
    b[0]++;
    b[1]++;
}
```

```fortran
INTEGER :: a = 123
INTEGER, DIMENSION(0:1) :: b := (/456, 789/)
!$OMP TARGET
    a *= 2
    b(0) = b(0) + 1
    b(1) = b(1) + 1
!$OMP END TARGET
```

Implicit mapping - GPUs

```c
int a = 123;
int b[2] = {456, 789};
#pragma omp target firstprivate(a) shared(b)
{
    a *= 2;
    b[0]++;
    b[1]++;
}
```

```fortran
INTEGER :: a = 123
INTEGER, DIMENSION(0:1) :: b := (/456, 789/)
!$OMP TARGET FIRSTPRIVATE(a) SHARED(b)
    a *= 2
    b(0) = b(0) + 1
    b(1) = b(1) + 1
!$OMP END TARGET
```

Implicit mapping

By default:

- scalar variables are implicitly mapped as `firstprivate`[4][5].
- statically allocated array variables are implicitly mapped as `shared`.

[4] See Section 2.19.7 in OpenMP standard version 5.0

[5] Before OpenMP 4.5, it was `shared`.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.
- Transferring data to / from devices is slow, minimising them is therefore a good idea performance wise.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.
- Transferring data to / from devices is slow, minimising them is therefore a good idea performance wise.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.
- Transferring data to / from devices is slow, minimising them is therefore a good idea performance wise.

  `private` Uninitialised copies.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.
- Transferring data to / from devices is slow, minimising them is therefore a good idea performance wise.

`private`	Uninitialised copies.
`firstprivate`	Initialises copies upon entry.

Data movements

- Passing copies back and forth requires data movements between the host and the target(s).
- Can set everything as `shared` and be done with it.
- Transferring data to / from devices is slow, minimising them is therefore a good idea performance wise.

`private`	Uninitialised copies.
`firstprivate`	Initialises copies upon entry.
`lastprivate`	Initialises original variable upon exit to semantically last value in construct.

Explicit mapping: map clause

```
#pragma omp target map([[map-type-modifier[,][map-type-
    modifier[,]...]map-type : ]locator-list)
```

```
!$OMP TARGET MAP([[map-type-modifier[,][map-type-
    modifier[,]...]map-type : ]locator-list)
!$OMP END TARGET
```

- to
- from
- tofrom

Explicit mapping: `map` clause

```
#pragma omp target map([[map-type-modifier[,][map-type-
    modifier[,]...]map-type : ]locator-list)
```

```
!$OMP TARGET MAP([[map-type-modifier[,][map-type-
    modifier[,]...]map-type : ]locator-list)
!$OMP END TARGET
```

- `to`
- `from`
- `tofrom`
- `alloc`
- `delete`

`map(tofrom:my_var)` clause value

`map(tofrom:my_var)` clause value

- Value of data before target is sent to the device for the target region.

`map(tofrom:my_var)` clause value

- Value of data before target is sent to the device for the target region.
- Final value of the variable in target region brought back to the device.

`map(to:my_var)` clause value

- This clause value is used when passing variables to target constructs that need a access to a host-initialised variable, and/or whose updates done on the device are not needed back on the host.
- The variable `my_var` begins the target region with the value it had before entering the region.
- The value of variable `my_var` will be left unchanged on the host after the target region.

map(to:my_var) clause value

```c
int a = 123;
int b;
#pragma omp target map(to:a,b)
{
    a *= 2;
    b = 100;
}
```

```fortran
INTEGER :: a := 123
INTEGER :: b
!$OMP TARGET MAP(to:a,b)
    a *= 2;
    b = 100;
!$OMP END TARGET
```

`map(from:my_var)` clause value

- This clause value is used when passing variables to target constructs that will be initialised or update on the device, and the final value is needed back on the host.
- The variable `my_var` begins the target region with an uninitialised value.
- The value of variable `my_var` leaves the target region with the final value it had at the end of it.

map(from:my_var) clause value

```c
int a;
int b = 123;
#pragma omp target map(from:a, b)
{
    a = 123;
}
```

```fortran
INTEGER :: a
INTEGER :: b := 123
!$OMP TARGET MAP(from:a, b)
    a = 123
!$OMP END TARGET
```

Mapping of arrays

```c
int a* = (int*)malloc(sizeof(int) * 10);
#pragma omp target map(tofrom:a)
{
    a[5] = 123;
}
```

```fortran
INTEGER, ALLOCATABLE :: a
ALLOCATE(a(0:9))
!$OMP TARGET MAP(tofrom:a)
    a(5) = 123
!$OMP END TARGET
```

Mapping of arrays

- You can also map array variables.
- The notation requires to pass the interval of the array you want to map.[6]
- First, you pass the index of the first element to map.
- Second, you pass the **length** of the section to map.

Caution

You do not pass the start index and the end index, but the start index and the length of the array section to map.

[6] Because you can map a part of the array if you so wish.

Time to practise: `3.DataMapping`

Update the source code provided to minimise the data movements of the different variables involved in calculations.

Tips

You will need:

- the `target` construct
- the `to` clause
- the `fromto` clause
- the `from` clause

Table of Contents

`target data`

- The `target data` construct creates a data environment, with variables that will persist throughout the `target data` region.
- In this data environment, we can declare the mapping of variables once, to avoid redundant data movements between `target` constructs.
- `target` constructs enclosed in a `target data` construct inherit their data environment.

target data construct

```
int a = 0;
// Data moves to the device, stays until the end.
#pragma omp target data map(tofrom:a)
{
    // Some host code
    #pragma omp target
    {
        a = 45;
    }
    printf("a = %a\n", a);
    #pragma omp target
    {
        a++;
    }
    // Some host code
}
printf("a = %a\n", a);
```

target data construct

```fortran
INTEGER :: a := 0
! Data moves to the device, stays until the end.
!$OMP TARGET DATA map(tofrom:a)
    ! Some host code
    !$OMP TARGET
        a = 45
    !$OMP END TARGET
    WRITE(*, '(A,I0)') "a = ", a
    !$OMP TARGET
        a = a + 1
    !$OMP END TARGET
    ! Some host code
!$OMP END TARGET DATA
WRITE(*, '(A,I0)') "a = ", a
```

Arbitrary data environment management points

- Sometimes, we may want to use data environment in a less structured way.
- Typically happens when ones starts to have function calls, and potential orphane constructs.
- We may have at some point one variable that we know we need to pass to the device.

Arbitrary data environment management points

```c
#pragma omp target enter data map(alloc:a)
// Any code
#pragma omp target exit data map(delete:a)
```

```fortran
!$OMP TARGET ENTER DARA MAP(alloc:a)
// Any code
!$OMP TARGET EXIT DATA MAP(delete:a)
```

Note:

- for `target enter data`, it maps data to the device, clause can be `to` or `alloc` only.
- for `target exit data`, it unmaps data from the device, clause can be `from` or `delete` only.
- `alloc` and `delete` allow to manipulate data on the device without requiring data movements.

`map(alloc:my_var)` clause value

- Allocates a variable on the device, does not issue any data movement between the host and the device.

map(alloc:my_var) clause value

```c
int a = 123;
#pragma omp target enter data map(alloc:a)
#pragma omp target
{
    a = 246;
    // Something with A.
}
```

```fortran
INTEGER :: a := 123
!$OMP TARGET ENTER DATA MAP(alloc:a)
!$OMP TARGET
    a = 246
    ! Something with A.
!$OMP END TARGET
```

`map(delete:my_var)` clause value

- Deletes a variable on the device, does not issue any data movement between the host and the device.

map(delete:my_var) clause value

```c
// Some code...
#pragma omp target exit data map(delete:a)
```

```fortran
! Some code...
!$OMP TARGET EXIT DATA MAP(delete:a)
```

`target update` construct

Sometimes, we may want to get the current value of a variable mapped to a device.

```
int a;
#pragma omp target data map(from:a)
{
    #pragma omp target
    {
        a = 123;
    }
    printf("a = %d\n", a);
    #pragma omp target
    {
        a++;
    }
}
```

`target update` construct

Sometimes, we may want to get the current value of a variable mapped to a device.

```
int a;
#pragma omp target data map(from:a)
{
    #pragma omp target
    {
        a = 123;
    }
    #pragma omp target update from(a)
    printf("a = %d\n", a);
    #pragma omp target
    {
        a++;
    }
}
```

target update construct

Conversely, we may want to update the value of a variable mapped to a device.

```c
int a;
int b;
#pragma omp target data map(to:a) map(from:b)
{
    #pragma omp target
    {
        b = a;
    }
    a++;
    #pragma omp target
    {
        b = a;
    }
    printf("a = %d\n", a);
}
```

target update construct

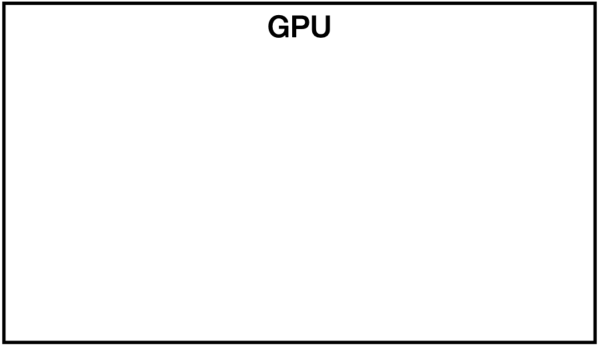
```
int a = 123;
#pragma omp target data map(to:a) map(from:result)
{
    #pragma omp target
    {
        result +=  a * 100;
    }
    a++;
    #pragma omp target update to(a)
    #pragma omp target
    {
        result +=  a * 100;
    }
}
```

Time to practise: `4.DataEnvironment`

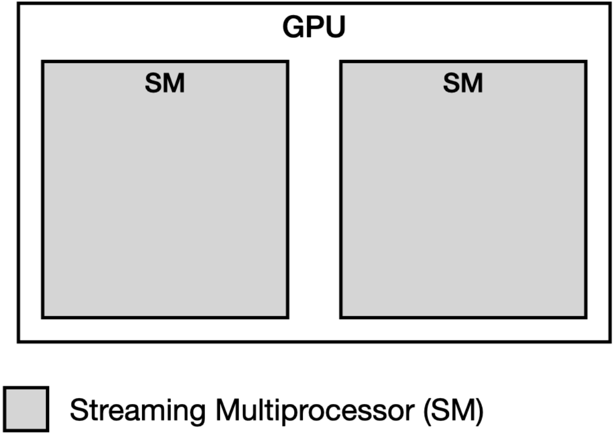
- Can ask for a target construct to be executed on a specific device.
- Try to get 2 GPUs, and get a target construct executed on each.
- Also try to get a target construct executed on the host device.
- Also try to get a target construct executed on a device that does not exist.

Table of Contents

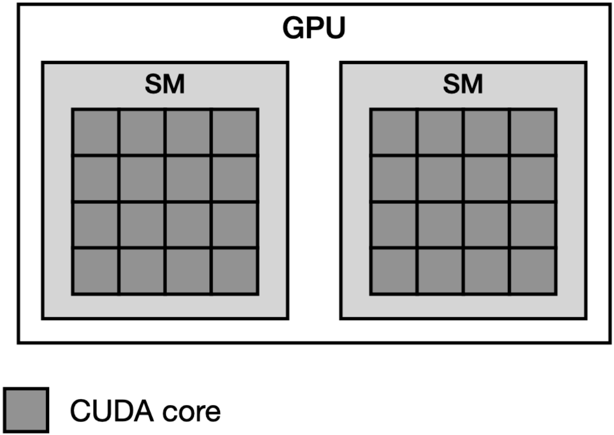
Hardware



Hardware



Hardware



`teams`

- The `teams` construct forks the execution into multiple teams of threads.

`teams`

- The `teams` construct forks the execution into multiple teams of threads.
- The code in the associated structured block will be executed by the master thread of each team.

`teams`

- The `teams` construct forks the execution into multiple teams of threads.
- The code in the associated structured block will be executed by the master thread of each team.
- Each team will therefore has its own "thread 0".

teams

Similarly to CPU OpenMP, you can:

- get your team number, using the `omp_get_team_num()` function.
- get the total number of teams, using the `omp_get_num_teams()` function.

teams

```c
#pragma omp target
{
    #pragma omp teams
    {
        int my_team_number = omp_get_team_num();
        printf("I am the master thread for team %d.\n",
    my_team_number);
    }
}
```

```fortran
!$OMP TARGET
    !$OMP TEAMS
        int my_team_number = omp_get_team_num();
        printf("I am the master thread for team %d.\n",
    my_team_number);
    !$OMP END TEAMS
!$OMP END TARGET
```

Composite `target teams` construct

```c
#pragma omp target teams
{
    int my_team_number = omp_get_team_num();
    printf("I am the master thread for team %d.\n",
   my_team_number);
}
```

```fortran
!$OMP TARGET TEAMS
    int my_team_number = omp_get_team_num();
    printf("I am the master thread for team %d.\n",
   my_team_number);
!$OMP END TARGET TEAMS
```

`teams`

- Similarly to the `parallel` construct, you can control the number of teams spawned.

`teams`

- Similarly to the `parallel` construct, you can control the number of teams spawned.
- Use the `num_teams` clause.

teams

- Similarly to the `parallel` construct, you can control the number of teams spawned.
- Use the `num_teams` clause.
- When used, this clause indicates that **at most** <X> teams will be spawned.

teams

```c
#pragma omp target teams num_teams(X)
{
    int my_team_number = omp_get_team_num();
    printf("I am the master thread for team %d.\n",
   my_team_number);
}
```

```fortran
!$OMP TARGET NUM_TEAMS(X)
    int my_team_number = omp_get_team_num();
    printf("I am the master thread for team %d.\n",
   my_team_number);
!$OMP END TARGET
```

`teams`

- Inside your `teams` construct, by default you just have the master thread running, just like in classic serial program execution.

`teams`

- Inside your `teams` construct, by default you just have the master thread running, just like in classic serial program execution.
- When wanting to fork threads in, you can use the `parallel` construct.

teams

```
#pragma omp target
{
    // One team of one thread
    #pragma omp teams
    {
        // Multiple teams of one thread
        #pragma omp parallel
        {
            // Multiple teams of multiple threads
        }
    }
}
```

Loops!

```c
for(int i = 0; i < N; i++)
{
    a[i] = i + 123;
}
```

```fortran
DO i = 0, N - 1
    a(i) = i + 123
END DO
```

Loops!

```c
#pragma omp target teams
{
    for(int i = 0; i < N; i++)
    {
        a[i] = i + 123;
    }
}
```

```fortran
!$OMP TARGET TEAMS
    DO i = 0, N - 1
        a(i) = i + 123
    END DO
!$OMP END TARGET TEAMS
```

Multi-level parallelism

- Just like in a `parallel` construct, threads do not automatically split iterations unless they are asked to.

Multi-level parallelism

- Just like in a `parallel` construct, threads do not automatically split iterations unless they are asked to.
- The `distribute` clause is equivalent to the `for` / `do` clause for CPUs.

Multi-level parallelism

- Just like in a `parallel` construct, threads do not automatically split iterations unless they are asked to.
- The `distribute` clause is equivalent to the `for` / `do` clause for CPUs.
- It allows the iteration set to be split across the master threads of all teams.

Loops!

```c
#pragma omp target teams
{
    #pragma omp distribute
    for(int i = 0; i < N; i++)
    {
        a[i] = i + 123;
    }
}
```

```fortran
!$OMP TARGET TEAMS
    !$OMP DISTRIBUTE
    DO i = 0, N - 1
        a(i) = i + 123
    END DO
!$OMP END TARGET TEAMS
```

Composite `target teams distribute` construct

```c
#pragma omp target teams distribute
for(int i = 0; i < N; i++)
{
    a[i] = i + 123;
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE
DO i = 0, N - 1
    a(i) = i + 123
END DO
```

Scheduling

- On CPUs, the `schedule` clause is available.

Scheduling

- On CPUs, the `schedule` clause is available.
- On GPUs, the `dist_schedule` clause is available.

Scheduling

- On CPUs, the `schedule` clause is available.
- On GPUs, the `dist_schedule` clause is available.
- Only `static` schedule available however.

Scheduling

- On CPUs, the `schedule` clause is available.
- On GPUs, the `dist_schedule` clause is available.
- Only `static` schedule available however.
- Can still specify a particular `chunksize`.

Scheduling

```c
#pragma omp target teams distribute
for(int i = 0; i < N; i++)
{
    a[i] = i + 123;
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE
DO i = 0, N - 1
    a(i) = i + 123
END DO
```

Scheduling

```c
#pragma omp target teams distribute schedule(static, 24)
for(int i = 0; i < N; i++)
{
    a[i] = i + 123;
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE SCHEDULE(STATIC, 24)
DO i = 0, N - 1
    a(i) = i + 123
END DO
```

Nested loops

```c
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < N; j++)
    {
        a[i][j] = i * j + 123;
    }
}
```

```fortran
DO i = 0, N - 1
    DO j = 0, j < N - 1
        a(i,j) = i * j + 123
    END DO
END DO
```

Nested loops

```c
#pragma omp target teams distribute
for(int i = 0; i < N; i++)
{
    for(int j = 0; j < N; j++)
    {
        a[i][j] = i * j + 123;
    }
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE
DO i = 0, N - 1
    DO j = 0, j < N - 1
        a(i,j) = i * j + 123
    END DO
END DO
```

Nested loops

```c
for(int i = 0; i < N; i++)
{
    #pragma omp target teams distribute
    for(int j = 0; j < N; j++)
    {
        a[i][j] = i * j + 123;
    }
}
```

```fortran
DO i = 0, N - 1
    !$OMP TARGET TEAMS DISTRIBUTE
    DO j = 0, j < N - 1
        a(i,j) = i * j + 123
    END DO
END DO
```

Nested loops

```c
#pragma omp target teams distribute
for(int i = 0; i < N; i++)
{
    #pragma omp parallel for
    for(int j = 0; j < N; j++)
    {
        a[i][j] = i * j + 123;
    }
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE
DO i = 0, N - 1
    !$OMP PARALLEL DO
    DO j = 0, j < N - 1
        a(i,j) = i * j + 123
    END DO
END DO
```

Massive loop

```
for(int i = 0; i < SUPER_BIG_N; i++)
{
    a[i = i + 123;
}
```

```
DO i = 0, SUPER_BIG_N - 1
    a(i) = i + 123
END DO
```

Massive loop

```c
#pragma omp target teams distribute parallel for
for(int i = 0; i < SUPER_BIG_N; i++)
{
    a[i = i + 123;
}
```

```fortran
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
DO i = 0, SUPER_BIG_N - 1
    a(i) = i + 123
END DO
```

Table of Contents

Multi-GPU

- Nowadays, GPU nodes contain more than 1 GPU.

Multi-GPU

- Nowadays, GPU nodes contain more than 1 GPU.
- On Bridges2, GPU nodes have 8 GPUs.

Multi-GPU

- Nowadays, GPU nodes contain more than 1 GPU.
- On Bridges2, GPU nodes have 8 GPUs.
- Fully using a GPU is great.

Multi-GPU

- Nowadays, GPU nodes contain more than 1 GPU.
- On Bridges2, GPU nodes have 8 GPUs.
- Fully using a GPU is great.
- Fully using all GPUs is more fun.

Find our way in a multi-GPU environment.

Find our way in a multi-GPU environment.

- Implicitly, one of the devices is selected by default.

Find our way in a multi-GPU environment.

- Implicitly, one of the devices is selected by default.
- You can know which one, by calling `omp_get_default_device()`.

Find our way in a multi-GPU environment.

- Implicitly, one of the devices is selected by default.
- You can know which one, by calling `omp_get_default_device()`.
- You can change which one, by calling `omp_set_default_device(new_device)`.

Devices number and identifiers

[7] Does not seem to be supported yet, as of `nvhpc/22.2`.

Devices number and identifiers

- The host is a device too, you can get its number by calling `omp_get_initial_device()`.

[7] Does not seem to be supported yet, as of `nvhpc/22.2`.

Devices number and identifiers

- The host is a device too, you can get its number by calling `omp_get_initial_device()`.
- You can get the number of the device you are currently running on by calling `omp_get_device_num()`.[7]

[7] Does not seem to be supported yet, as of `nvhpc/22.2`.

Devices number and identifiers

- The host is a device too, you can get its number by calling `omp_get_initial_device()`.
- You can get the number of the device you are currently running on by calling `omp_get_device_num()`.[7]
- You can get the total number of target devices by calling `omp_get_num_devices()`.

[7] Does not seem to be supported yet, as of `nvhpc/22.2`.

Devices number and identifiers

- The host is a device too, you can get its number by calling `omp_get_initial_device()`.
- You can get the number of the device you are currently running on by calling `omp_get_device_num()`.[7]
- You can get the total number of target devices by calling `omp_get_num_devices()`.

Note

`omp_get_device_num()` returns the number of **target devices**, it does **not** include the host device.

[7] Does not seem to be supported yet, as of `nvhpc/22.2`.

Splitting the work

- Send different parts of an array to different GPUs.
- Have multiple GPUs work in parallel on a given arrray.

Addressing GPUs - The `device` clause

- Can ask for a target construct to be executed on a specific device.
- Just use the `device` clause on a `target` construct.

```c
// This will execute on the first GPU, device #0.
#pragma omp target device(0)
{
    // Code to execute on device 0
}
```

```fortran
! This will execute on the first GPU, device #0.
!$OMP TARGET DEVICE(0)
    ! Code to execute on device 0
!$OMP END TARGET
```

Time to practise: 6.MultiGPU

- Can ask for a target construct to be executed on a specific device.
- Try to get 2 GPUs, and get a target construct executed on each.
- Also try to get a target construct executed on the host device.
- Also try to get a target construct executed on a device that does not exist.

Table of Contents

Asynchronous

- By default, target constructs are synchronous.
- You cannot process beyond it until it is finished.
- The `nowait` clause allows to execute a target construct in an asynchronous fashion.
- Execution does not wait for the end of the target construct to continue its progress.
- Dependencies between targets are expressed using the `depend` clause.
- If a `target` construct with a `nowait` clause does not have `depend` clauses, it is assumed to have no dependencies and can execute freely at any time.

Asynchronous

`depend(in:a)` I will read from variable `a`.

`depend(out:a)` I will write to variable `a`.

`depend(inout:a)` I will read from, and write to, variable `a`.

Structured synchronisation

```
#pragma omp target nowait map(tofrom:a) depend(inout:a)
{
    a = 456;
}
#pragma omp target nowait map(to:a) depend(in:a)
{
    b = a + 3;
}
#pragma omp target map(to:a) map(from:c) \
                   depend(in:a) depend(out:c)
{
    c = a + 10
}
```

Structured synchronisation

```
!$OMP TARGET NOWAIT MAP(TOFROM:a) DEPEND(inout:a)
    a = 456;
!$OMP END TARGET
!$OMP TARGET NOWAIT MAP(TO:a) MAP(FROM:b) &
                        DEPEND(IN:a) DEPEND(OUT:b)
    b = a + 3;
!$OMP END TARGET
!$OMP TARGET MAP(FROM:a) MAP(TO:c) &
                 DEPEND(IN:a) DEPEND(OUT:c)
    c = a + 10
!$OMP END TARGET
```

Construct synchronisation[8]

```
#pragma omp single
{
    #pragma omp target nowait map(tofrom:a) depend(inout:a)
    {
        a = 456;
    }
    #pragma omp target nowait map(to:a) depend(in:a)
    {
        b = a + 3;
    }
    #pragma omp target nowait map(to:a) map(from:c) \
                       depend(in:a) depend(out:c)
    {
        c = a + 10
    }
}
```

Construct synchronisation[9]

```
!$OMP SINGLE
!$OMP TARGET NOWAIT MAP(TOFROM:a) DEPEND(inout:a)
    a = 456;
!$OMP END TARGET
!$OMP TARGET NOWAIT MAP(TO:a) MAP(FROM:b) &
                    DEPEND(IN:a) DEPEND(OUT:b)
    b = a + 3;
!$OMP END TARGET
!$OMP TARGET NOWAIT MAP(FROM:a) MAP(TO:c) &
             DEPEND(IN:a) DEPEND(OUT:c)
    c = a + 10
!$OMP END TARGET
!$OMP END SINGLE
```

[9] Example assumes everything in `parallel` construct

Explicit synchronisation

```
#pragma omp target nowait map(tofrom:a) depend(inout:a)
{
    a = 456;
}
#pragma omp target nowait map(to:a) depend(in:a)
{
    b = a + 3;
}
#pragma omp target nowait map(to:a) map(from:c) \
                   depend(in:a) depend(out:c)
{
    c = a + 10
}
#pragma omp taskwait
```

Construct synchronisation[10]

```
!$OMP TARGET NOWAIT MAP(TOFROM:a) DEPEND(inout:a)
    a = 456;
!$OMP END TARGET
!$OMP TARGET NOWAIT MAP(TO:a) MAP(FROM:b) &
                    DEPEND(IN:a) DEPEND(OUT:b)
    b = a + 3;
!$OMP END TARGET
!$OMP TARGET NOWAIT MAP(FROM:a) MAP(TO:c) &
             DEPEND(IN:a) DEPEND(OUT:c)
    c = a + 10
!$OMP END TARGET
!$OMP TASKWAIT
```

[10] Example assumes everything in `parallel` construct

Time to practise: `7.Asynchronous`

Execute the different statements shown in different `target` constructs, using asynchronous execution.

Tips

You will need:

- `target` construct
- `nowait` clause
- `depend` clause
- `map` clause

Table of Contents

Summary

You now know how to:

- offload your workload to GPUs.
- control the mapping of data between host and targets.
- handle multiple GPUs.
- issue kernels in asynchronous fashion.

To be seen

- Reverse offloading
- Coalescing memory accesses.
- Metadirectives
- Directive cancellation
- Conditional execution
- And a lot, lot more...

To be seen

- Reverse offloading
- Coalescing memory accesses.
- Metadirectives
- Directive cancellation
- Conditional execution
- And a lot, lot more...

The best place to learn more about OpenMP and how it works, to get the specifications and so on is the OpenMP forum website.

Your opinion matters

Figure: Link to surveys