# Directive Based GPU Programming

Orian Louant

CÉCI-CISM Training – November 30 2023

# The zoo of programming model for accelerators

**Central Processing Unit (CPU)**

- latency-optimized
- general-purpose
- wide range of distinct tasks sequentially

**Graphics Processing Unit (GPU)**

- throughput-optimized
- specialized
- highly parallel computing

**Low-Level Languages**

- Cuda (NVIDIA)
- HIP (AMD)
- OpenCL (neutral)

**High-Level Frameworks**

- Kokkos
- Raja
- Alpaka
- SyCL (DPC++)

**Directive Based Models**

- OpenMP
- OpenACC

# Programming with directives

## OpenMP

- general-purpose parallel programming model
- the programmer explicitly spread the execution of loops, code regions, and tasks across team(s) of threads

## OpenACC

- oriented towards accelerators
- the programmer tells to the compiler which loops can be parallelized and let the compiler do the mapping to the target architecture

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


!$omp construct [clauses]
  code-block
!$omp end construct
```

```
#pragma acc construct [clauses]
  structured-block


!$acc construct [clauses]
  code-block
!$acc end construct
```

# OpenMP support for accelerator

- introduced with OpenMP 4.0, significantly extended in versions 4.5 and 5.0

- GPUs are the most common type of accelerator

- OpenMP is not limited to GPUs, you can use it to target any kind of accelerators (NEC SX-Aurora TSUBASA, FPGAs, ASICs, ...)

- makes it easier to target multiple heterogeneous architectures using the same code base
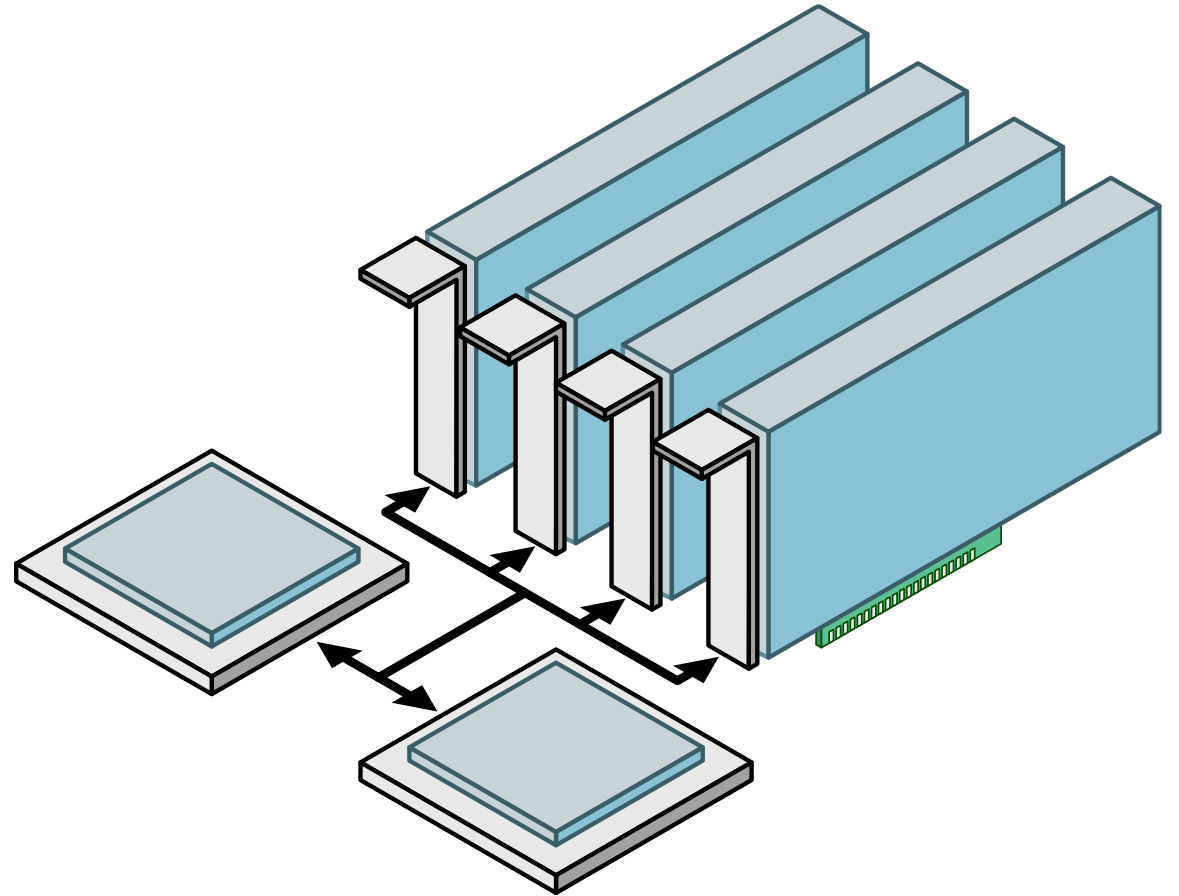
# OpenMP execution model
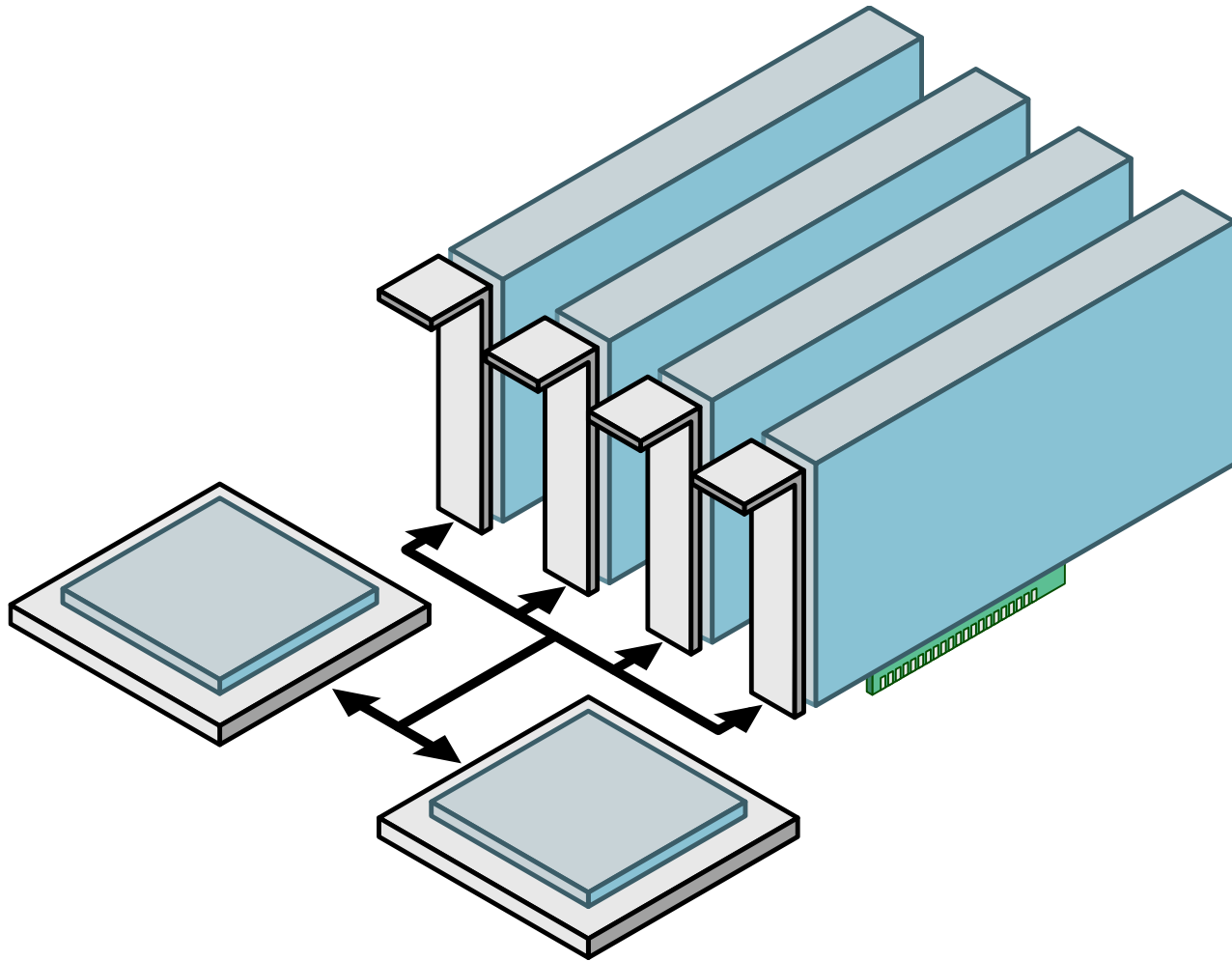


**Host**

Where the execution starts. In almost all cases, this is the CPU

**Device**

Multiple accelerator/coprocessor of the same type for offloading

# Host Multithreading

As a starting point to our journey to the world of GPU programming with directives, we will use a very simple kernel: `saxpy`

- `parallel:` create a team of threads that will start executing in parallel

- `for/do:` distribute the iteration of the loop within the team of threads

```c
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}
```

```fortran
!$omp parallel do
do i = 1,n
  y(i) = a * x(i) + y(i)
end do
!$omp end parallel do
```

# Offloading execution

The `target` directive instructs the compiler to generate a target task that will execute the enclosed block of code on a device

```
#pragma omp target
   structured-block
```
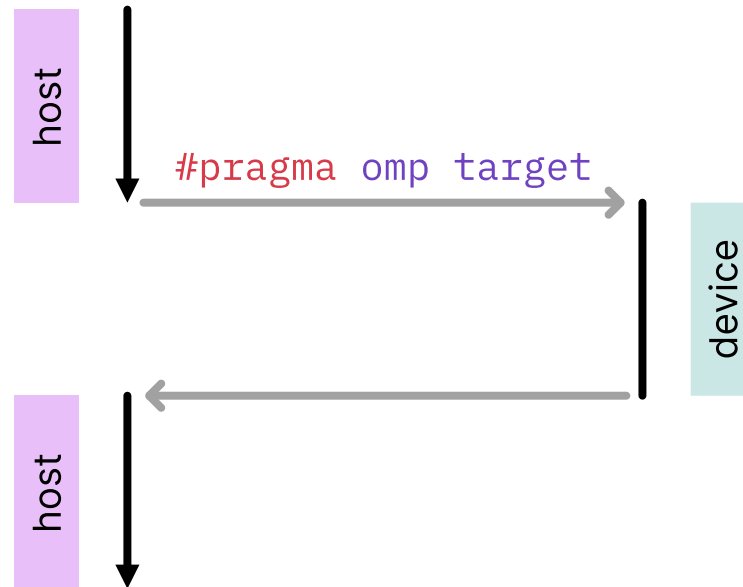
```
!$omp target
   code-block
!$omp end target
```



```
#pragma omp target parallel for
for (int i = 0; i < n; ++i) {
   y[i] = a * x[i] + y[i];
}
```

```
!$omp target parallel do
do i = 1,n
   y(i) = a * x(i) + y(i)
end do
!$omp end target parallel do
```

# Gather devices information

The `omp_get_num_devices` routine returns the number of target devices

Devices are assigned an ID from `0` to `ndevice-1`. You can select the device to use for a `target` region by using the `device` clause

The `omp_is_initial_device` routine returns `true` if the current task is executing on the host device (CPU). It returns `false` if this is not the case

```c
int on_host;
int ndev = omp_get_num_devices();

printf("Number of devices: %d\n", ndev);

for (int i = 0; i < ndev; i++) {
  #pragma omp target device(i) map(from:on_host)
  {
    on_host = omp_is_initial_device();
  }

  printf("Is initial device when on device %d: %d\n",
         i, on_host);
}

printf("Is initial device when on host: %d\n",
       omp_is_initial_device());
```

# Data in the device memory

Variable and arrays are present in the host (CPU) memory but not in the device memory

- in order to use a variable/array on the device, we need to have the data present in the device memory

- if we want to use data computed on the device, we need to update the data present in the host memory

```
#pragma omp target map(type:list)
   structured-block
```

```
!$omp target map(type:list)
   code-block
!$omp end target
```

| Type | Description |
|------|-------------|
| alloc | allocate memory on the device |
| to | allocate memory on the device and copy the original values from the host to the device |
| from | allocate memory on the device and copy the values from the device to the host |
| tofrom | combination of to and from type |

# Data in the device memory

- in C/C++, when moving array to and from the GPU, you need to specify the number of elements to be copied

- this is not required in Fortran

You can also copy part of an array:

```
#pragma omp target map(to:b[10:4])
```

```
!$omp target map(to:b[10:13])
```

**Note:** in C/C++ the syntax is [start:length] and [start:end] in Fortran

```c
double  a = 1234;
double *b = (double*)malloc(sizeof(double)*n);

#pragma omp target map(tofrom:a) \
                    map(to:b[0:n])
{
  // Code using a and b on the GPU
}
```

```fortran
real(kind=real64)                :: a
real(kind=real64), allocatable :: b(:)

allocate(b(n))

!$omp target map(tofrom:a) map(to:b)
  ! Code using a and b on the GPU
!$omp end target
```

# Moving data to and from the device

- `map(to:x):` because we only read the array on the device

- `map(tofrom:y):` because we read and modify the array on the device

Scalar variables that do not appear in a `map` clause default to `firstprivate`. As a consequence we don't need to map the variable `a` to the device

```
#pragma omp target parallel for map(to:x[0:n]) map(tofrom:y[0:n])
for (int i = 0; i < n; ++i) {
    y[i] = a * x[i] + y[i];
}




!$omp target parallel do map(to:x) map(tofrom:y)
do i = 1,n
    y(i) = a * x(i) + y(i)
end do
!$omp end target parallel do
```

# Compilers

## Clang (NVIDIA)

```
clang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda
      -Xopenmp-target=nvptx64-nvidia-cuda -march=<sm_XY> <source>
```

## Clang (AMD)

```
clang -fopenmp -fopenmp-target=amdgcn-amd-amdhsa
      -Xopenmp-target=amdgcn-amd-amdhsa –march=gfx<XXX> <source>
```

## NVIDIA HPC SDK (NVIDIA only)

```
nvc/nvfortran –mp=gpu –Minfo=mp –gpu=<ccXY> <source>
```

## GCC (NVIDIA, *ok performance with recent version*)

```
gcc/gfortran –fopenmp -foffload=nvptx-none <source>
```

# Using the Cray compiler (Lucia, LUMI)

Cray utilize target modules to define the target architecture for compilation. Once a `craype-accel-*` module is loaded, offloading is automatic if OpenMP is enabled (`-fopenmp`)

## Cray cc (NVIDIA)

```
module load PrgEnv-cray
module load nvhpc
module load craype-accel-nvidia80
cc –fopenmp <source>
```

## Cray cc (AMD)

```
module load PrgEnv-cray
module load rocm
module load craype-accel-amd-gfx90a
cc –fopenmp <source>
```
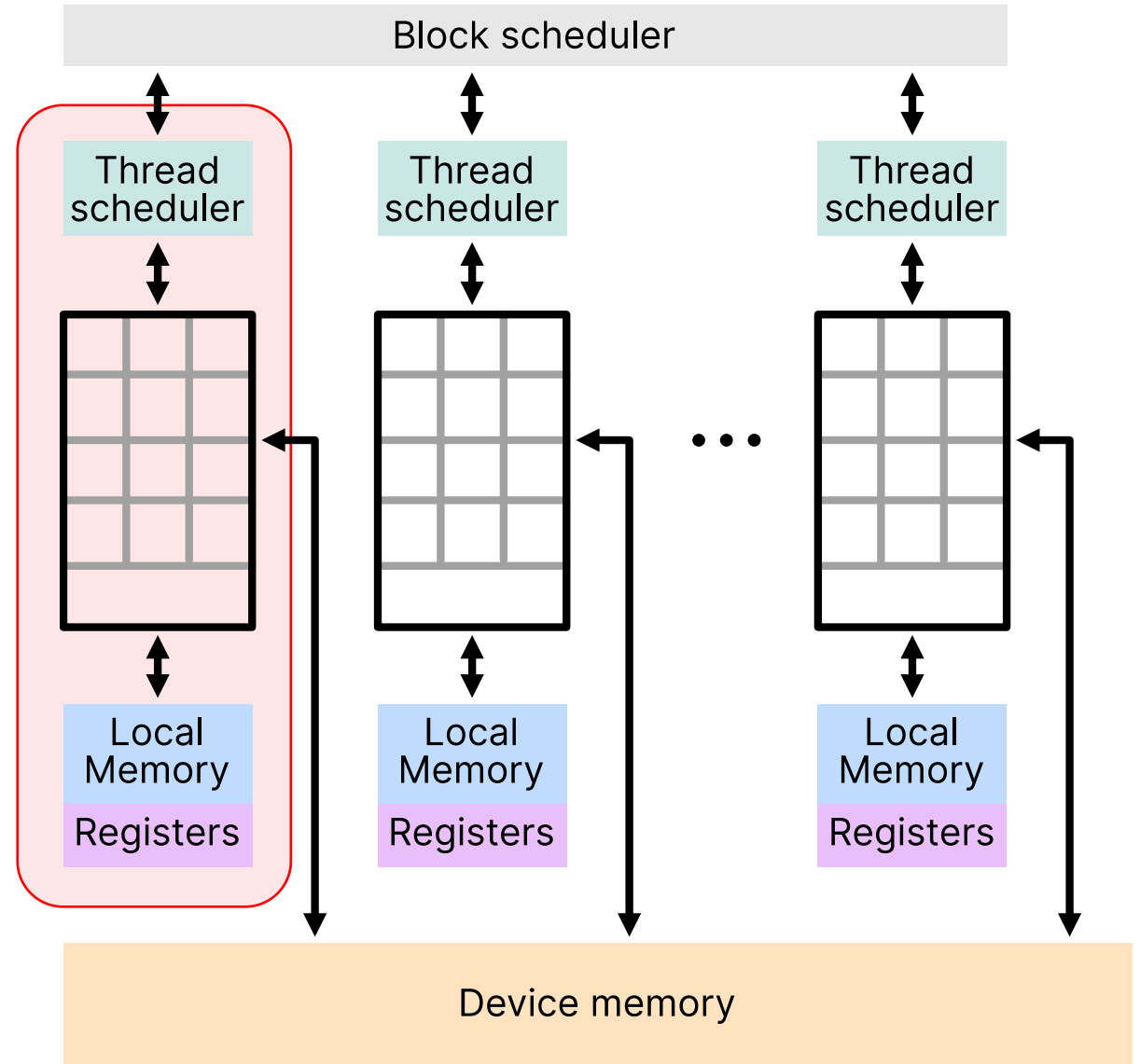
# A look to the hardware

A GPU is composed of multiple units each with their own registers, local memory and scheduler

- streaming multiprocessors (NVIDIA)
- compute units (AMD)

On a GPU, the work is scheduled in blocks that are executed on these units

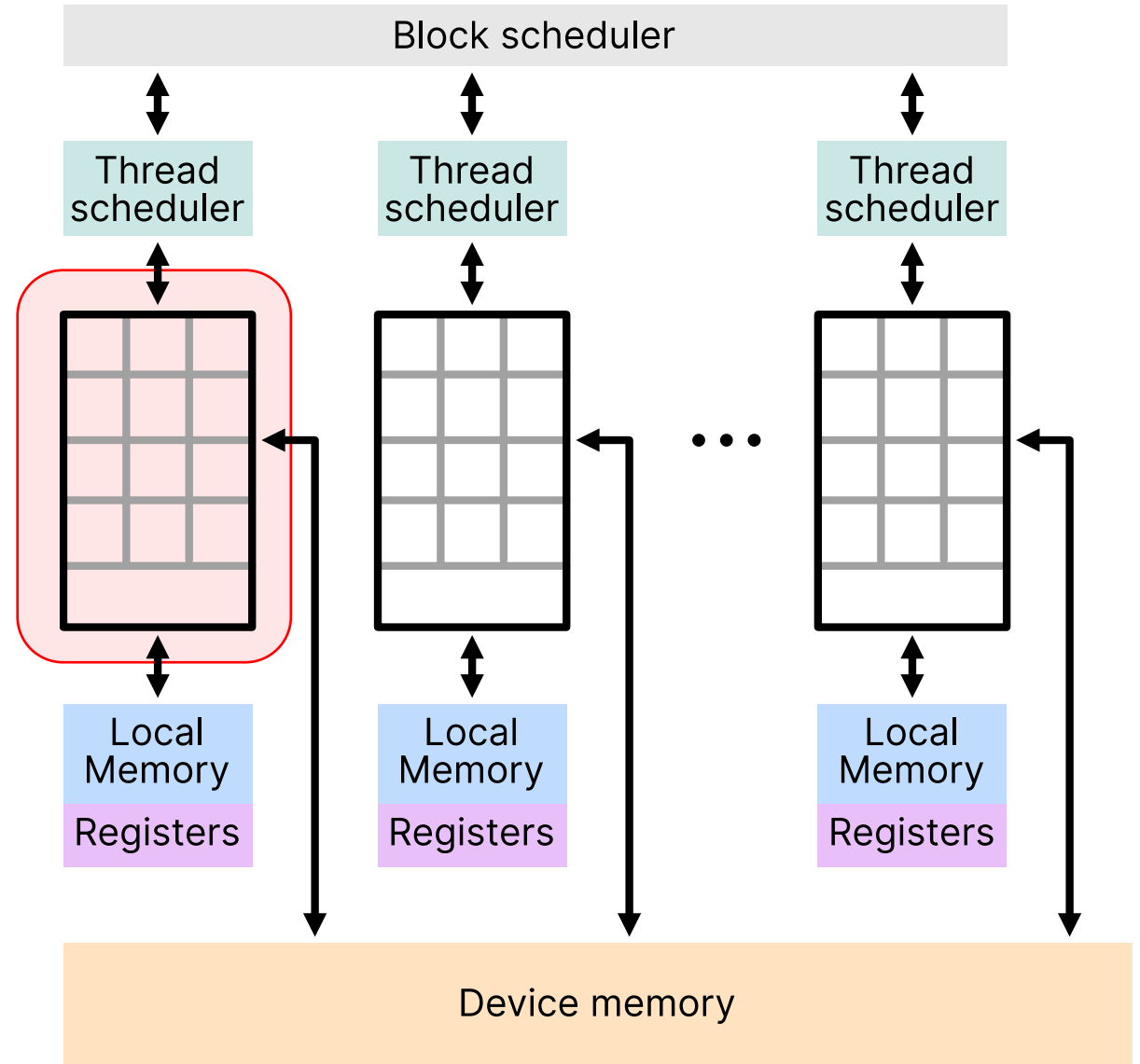- thread blocks (NVIDIA)
- workgroups (AMD)

# A look to the hardware

The threads in a block are further divided in bundles that execute in lockstep: they run the same instructions, and follow the same control-flow path (SIMD fashion)

- 32 threads: warps (NVIDIA)
- 64 workitems: wavefront (AMD)

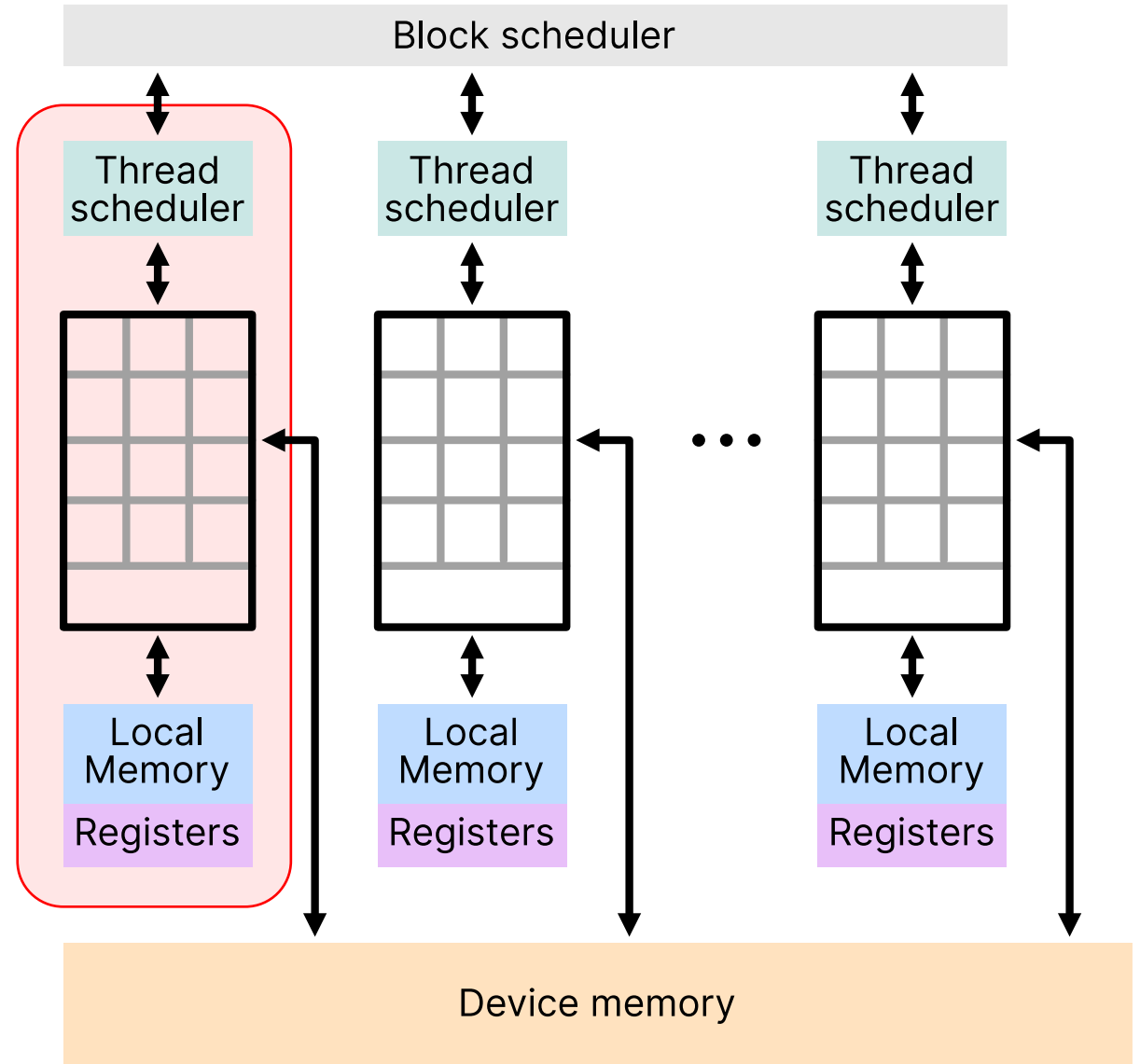These bundles of threads execute on the vector units of the GPU

# A look to the hardware

```
#pragma omp target parallel for
for (int i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}
```

We create only one team of threads that will use only one of the available units of our GPU

We need a way to create multiple teams so that we use the full potential of the hardware

# The team construct: motivation

Let's consider some limitations of the hardware:

- no synchronization or memory fences possible between the streaming multiprocessors/compute units
- unlike CPUs where there is cache coherency between the cores, there is no such coherence between the streaming multiprocessors/compute units of a GPU

These limitations of the hardware have consequence if you consider "normal" OpenMP:
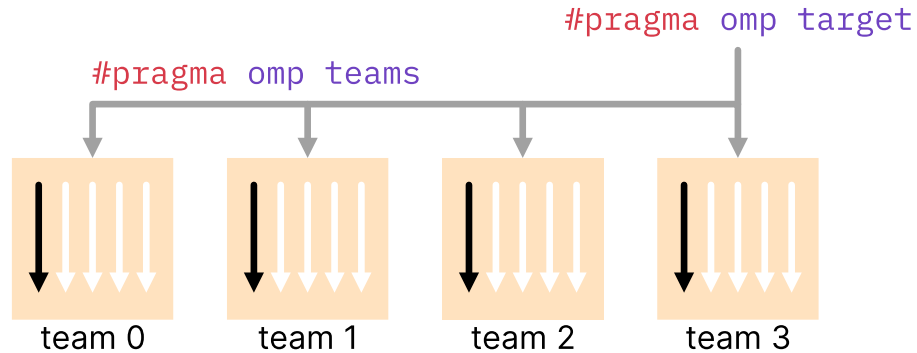
- creation of a parallel region, work-sharing tasks, ...
- barriers, critical regions, locks and atomics can be applied to a team of threads

In order to keep these characteristics on the devices an additional level was added, the `team` construct:
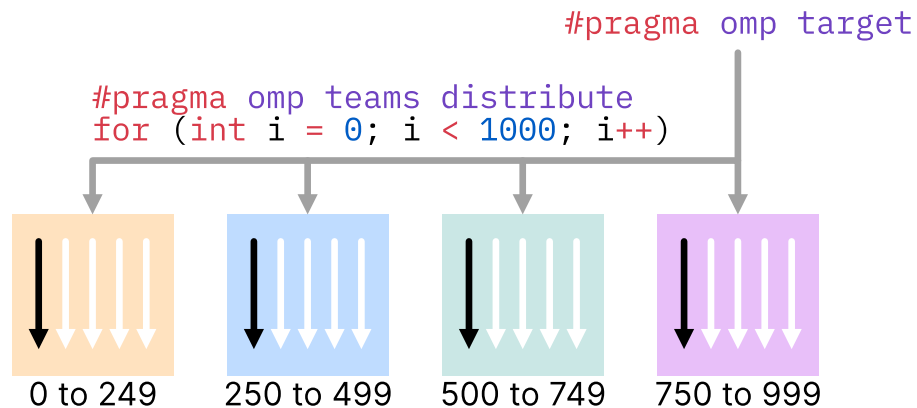
- multiple teams are spawned and each of these teams has a master threads
- the master thread can spawn a team of threads with a `parallel` construct
- threads in different teams cannot synchronize with each other but threads within a team can

# Creating teams and distribute work

When a teams construct is reached, a league of teams is created and the initial thread in each team executes the teams region

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

team 0    team 1    team 2    team 3

```
#pragma omp teams
    structured-block


!$omp teams
    structured-block
!$omp end teams
```

When a distribute construct is reached, the iterations of one or more loops will be distributed to the teams

```
#pragma omp target
#pragma omp teams distribute
for (int i = 0; i < 1000; i++)
```

0 to 249    250 to 499    500 to 749    750 to 999

```
#pragma omp distribute
    for-loops


!$omp distribute
    do-loops
!$omp end distribute
```

# Get teams information

The number of teams can be controlled by the `num_teams` clause and the number of threads with the `thread_limit` clauses

```
#pragma omp teams num_teams(nteams) \
                  thread_limit(nthreads)
```

```
!$omp  teams num_teams(nteams) &
!$omp&       thread_limits(nthreads)
```

In addition, OpenMP provide runtime functions:

- `omp_get_num_teams()` returns the number of teams
- `omp_get_team_num()` returns the team number of the calling threads (`0` to `nteams-1`)
- `omp_get_thread_limit()` returns the maximum number of threads

# Saxpy with teams distribute

- **target:** create a target task that will be executed on the GPU

- **team distribute:** create multiple teams of threads and distribute the loop iterations to these teams

- **parallel for/do:** distribute the iterations to the threads of the teams

```c
#pragma omp target teams distribute parallel for \
                 map(to:x[0:n]) map(tofrom:y[0:n])
for (int i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}
```

```fortran
!$omp target teams distribute parallel do map(to:x) map(tofrom:y)
do i = 1,n
  y(i) = a * x(i) + y(i)
end do
!$omp end target teams distribute parallel do
```

# Jacobi 2D – host version

$$u_{i,j}^{N+1} = \frac{1}{4}\left(u_{i+1,j}^N + u_{i-1,j}^N + u_{i,j+1}^N + u_{i,j-1}^N\right)$$
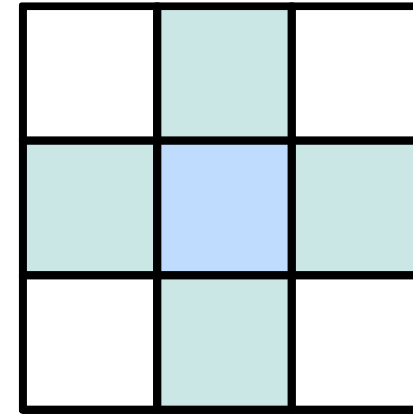
point we want to compute

additional points needed

```
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp parallel for reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                           +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // Swap values, uold <- unew
}
```
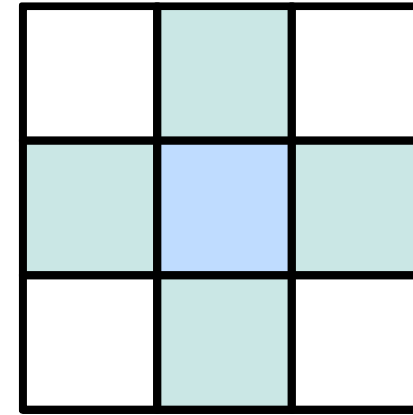
# Jacobi 2D – device version

$$u_{i,j}^{N+1} = \frac{1}{4}\left(u_{i+1,j}^N + u_{i-1,j}^N + u_{i,j+1}^N + u_{i,j-1}^N\right)$$



point we want to compute

additional points needed

```c
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp teams distribute parallel for reduction(max:err) \
                  map(tofrom:uold[0:n*m]) map(from:unew[0:n*m])
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                            +  uold[(j-1)*m +    i] + uold[(j+1)*m +    i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // Swap values, uold <- unew
}
```

# Jacobi 2D

- if we run the multithreaded version of the Jacobi code on a CPU, we get a good speedup up to 8 threads and close to 21x speedup when we use the entire socket (AMD EPYC 7542, 32 cores)

- if we run on the GPU we see a small speedup on AMD compared to the serial execution but 10x slower compared to the 32 threads run

- on NVIDIA, the performance is even worse, with a 0.89x speedup compared to the serial run

| Number of threads | Time (s) | Speedup |
|---|---|---|
| 1 | 28.433 | 1.00 |
| 4 | 7.140 | 3.98 |
| 8 | 3.718 | 7.64 |
| 16 | 2.117 | 13.43 |
| 32 | 1.376 | 20.66 |
| **AMD MI250x** [1] | 11.842 | 2.41 |
| **NVIDIA A100** [2] | 31.694 | 0.89 |

(1) Cray compiler, CPE 22.08, ROCm 5.1
(2) Clang 16.0.6, CUDA 11.7

# Jacobi 2D

In order to understand to poor performance of the GPU version, we will do a quick profiling

We can use `nsys` (NVIDIA) or the `LIBOMPTARGET_KERNEL_TRACE` environment variable (AMD).

```
__tgt_rtl_data_alloc:              64us
__tgt_rtl_data_alloc:              53us
__tgt_rtl_data_submit_async:       33674us
__tgt_rtl_data_alloc:              3us
__tgt_rtl_data_submit_async:       135us
__tgt_rtl_run_target_team_region:  4879us
__tgt_rtl_data_retrieve_async:     93us
__tgt_rtl_data_retrieve_async:     32358us
__tgt_rtl_data_retrieve_async:     32632us
__tgt_rtl_synchronize:             0us
__tgt_rtl_data_delete:             4us
__tgt_rtl_data_delete:             26us
__tgt_rtl_data_delete:             17us
```

| Number of threads | Time (s) | Speedup |
|---|---|---|
| 1 | 28.433 | 1.00 |
| 4 | 7.140 | 3.98 |
| 8 | 3.718 | 7.64 |
| 16 | 2.117 | 13.43 |
| 32 | 1.376 | 20.66 |
| **AMD MI250x** [1] | 11.842 | 2.41 |
| **NVIDIA A100** [2] | 31.694 | 0.89 |

(1) Cray compiler, CPE 22.08, ROCm 5.1
(2) Clang 16.0.6, CUDA 11.7

# Efficient movement of data

From the result of a quick profiling of the Jacobi code on the GPU, we see that

- moving data to and from the device at every iteration is inefficient

- better solution is to copy the data to the device and keep it on the device between iterations

For that we can use a structured data region that Map variables to a device data environment for the extent of the region

```
#pragma omp target data map(type:list)
    structured-block



!$omp target data map(type:list)
    structured-block
!$omp end target data
```

# Jacobi 2D with a structured data region

In order to improve the movement of data, we create a data region that covers the entire while loop so that we don't copy data to and from the GPU between iterations

```
#pragma omp target data map(tofrom:uold[0:n*m]) map(alloc:unew[0:n*m])
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp target teams distribute parallel for reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                           +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
} // end of the data region
```

# Unstructured data

- for data regions that span multiple lexical scopes (functions or files) you can use an unstructured data region

- data movement or allocation to the device is done with the `enter data` directive

- data movement or deallocation from the device is done with the `exit data` directive

- update of data in the middle of an unstructured data region, you can use the `target update` directive (from the host)

```
#pragma omp target enter data map(type:list)

#pragma omp target update to|from(list)

#pragma omp target exit data map(type:list)
```

```
!$omp target enter data map(type:list)

!$omp target update to|from(list)

!$omp end target exit data map(type:list)
```

# The update directive

- you can update data in the middle of a data region, you can use the `target update` directive with clauses

  - `from`: data on the host is updated with data from the device

  - `to`: data on the device is updated with the data from the host

- this directive can be used in the middle of a structured or unstructured data region

```
#pragma omp target data map(tofrom:a[0:n])
{
    // do something with a on the device
    #pragma omp target update from(a[0:n])
    // do something with a on the host
    #pragma omp target update to(a[0:n])
    // do something with a on the device
}
```

```
!$omp target data map(tofrom:a)
    ! do something with a on the device
    !$omp target update from(a)
    ! do something with a on the host
    !$omp target update to(a)
    ! do something with a on the device
!$omp end target data
```

# Jacobi 2D with unstructured data directives

```c
#pragma omp target enter data map(to:uold[0:n*m]) map(alloc:unew[0:n*m])

while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp target teams distribute parallel for reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                           +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
}

#pragma omp target exit data map(from:uold[0:n*m]) map(delete:unew[0:n*m])
```

# Jacobi 2D with a data region

Now that we have removed unnecessary data movement we see

- a huge improvement compared to the first version without data movement optimization

- still, we only achieved a ~9x speedup on the MI250x GPU compared to the serial CPU version

- Significant speed up for the A100 (28x)

| Number of threads | Time (s) | Speedup |
|---|---|---|
| 1 | 28.433 | 1.00 |
| 4 | 7.140 | 3.98 |
| 8 | 3.718 | 7.64 |
| 16 | 2.117 | 13.43 |
| 32 | 1.376 | 20.66 |
| **AMD MI250x** (1) | 11.842 | 2.41 |
| (2) | 3.265 | 8.71 |
| **NVIDIA A100** (1) | 31.694 | 0.89 |
| (2) | 1.010 | 28.15 |

(1) with no data movement optimization
(2) with data movement optimization

# Enabling more parallelism

By only parallelizing the outer loop, we do not fully exploit the parallelism of the hardware

- distribute the iterations of the outer loop to the teams
- distribute the iterations of the inner loop to the threads of the hardware

```
#pragma omp target data map(tofrom:uold[0:n*m]) map(alloc:unew[0:n*m])
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp target teams distribute reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    #pragma omp parallel for reduction(max:err)
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                            +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
} // end of the data region
```

# Enabling more parallelism

By distributing both loops, the first across teams and the second across threads we increase parallelism.

- for CPUs it's not recommended to use more threads than the available cores/hardware threads on the system

- for GPUs, in order to hide memory latency, you need to use more threads than what the hardware is capable of executing at the same time

- Increasing the parallelism leads to 2x improvement on the A100 and more than 6x for the MI250x compared to the version where we only parallelize the outer loop

| Number of threads | Time (s) | Speedup |
|---|---|---|
| 1 | 28.433 | 1.00 |
| 4 | 7.140 | 3.98 |
| 8 | 3.718 | 7.64 |
| 16 | 2.117 | 13.43 |
| 32 | 1.376 | 20.66 |
| **AMD MI250x** (1) | 3.265 | 8.71 |
| (2) | 0.526 | 54.05 |
| **NVIDIA A100** (1) | 1.010 | 28.15 |
| (2) | 0.581 | 48.94 |

(1) parallelization of the outer loop
(2) outer loop across teams and inner loop across threads

# Enabling more parallelism with loop collapsing

Another way to increase the parallelism is to collapse the loop nest. For a `for` or `distribute` construct, if a collapse clause is present and more the one loop is associated with the construct, then the iteration of all associated loops are collapsed into one larger iteration space

```c
#pragma omp target data map(tofrom:uold[0:n*m]) map(alloc:unew[0:n*m])
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma omp target teams distribute parallel for collapse(2) reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                           +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
} // end of the data region
```

# Loop collapsing

Collapsing the loops is an other way to increase parallelism on the GPU

- No change is the observed speedup for the MI250x as well as for the A100

| Number of threads | Time (s) | Speedup |
|---|---|---|
| 1 | 28.433 | 1.00 |
| 4 | 7.140 | 3.98 |
| 8 | 3.718 | 7.64 |
| 16 | 2.117 | 13.43 |
| 32 | 1.376 | 20.66 |
| **AMD MI250x** (1) | 0.526 | 54.05 |
| (2) | 0.522 | 54.46 |
| **NVIDIA A100** (1) | 0.566 | 48.94 |
| (2) | 0.581 | 48.93 |

(1) outer loop across teams and inner loop across threads
(2) collapsing the two loops

# Structure mapping

Scalar member of a structure will be copied at the same time as the base structure

However, OpenMP target offload do not have support for deep copy:

- we have to be careful when using structure with non-scalar members (arrays)

- non-scalar members need to be mapped explicitly by the developer

Structure pointers are arrays of size 1 and need to be mapped explicitly by the developer

```c
typedef struct saxpy_data {
  int size;
  float *x, *y;
  float a;
} saxpy_data_t;

void saxpy(saxpy_data_t *data) {
  #pragma omp target teams distribute parallel for     \
                      map(    to:data[0:1])            \
                      map(    to:data->x[0:data->size]) \
                      map(tofrom:data->y[0:data->size])
  for(int i = 0; i < data->size; i++) {
    data->y[i] = data->a * data->x[i] + data->y[i];
  }
}
```

# User-defined mapper

When a structure is heavily used in the application, it can be tiresome to explicitly map it every time we want to use it to the device

OpenMP offer user-defined mapper to tackle this problem: the developer can define how a structure should be mapped to the device

The `declare mapper` directive declares a user-defined mapper for a given type and may define a `mapper-identifier` that can be used in a map clause.

```
#pragma omp declare mapper([mapper-identifier:]type var) \
                          [clause[ [,] clause] ... ]
```

where `clause` is `map([map-type: ] list)`which is the same syntax as for the mapping clause of a data region.

# User-defined mapper

Using the `declare mapper` directive, we can how a structure should be copied to the device:

- we specify that the mapper applies to the `saxpy_data_t` type and defines an identifier for the structure (`data`)

- we map the base structure with a `to` modifier which means that the `size` and `a` members will be copied to the device when entering the data region but not copied back the host when we exit the region. We also allocate and attach the x and y arrays

- as we use a pointer to the `saxpy_data_t` structure, the mapping of the structure to the device is done with `map(data[0:1])`

```
typedef struct saxpy_data {
  int size;
  float *x, *y;
  float a;
} saxpy_data_t;

#pragma omp declare mapper(saxpy_data_t data)   \
                map(to:data)                     \
                map(to:data.x[0:data.size]) \
                map(   data.y[0:data.size])

void saxpy(saxpy_data_t *data) {
  #pragma omp target teams distribute parallel for \
                map(data[0:1])
  for(int i = 0; i < data->size; i++) {
    data->y[i] = data->a * data->x[i] + data->y[i];
  }
}
```

# Prescriptive vs. Descriptive model

**OpenMP is descriptive**

The OpenMP directives instruct the compiler to generate parallel code in a specific way, leaving little to the discretion of the compiler

**OpenACC is prescriptive**

OpenACC directives tell the compiler that a loop can be parallelized. The compiler is free to run them in parallel any way it chooses and choose very different mappings depending on the underlying hardware

# The kernels construct

The kernels construct defines a region that is to be compiled into a sequence of kernels for execution on the device

- the compiler will analyse the code and splits it in a sequence of device kernels
- typically each loop nest will become a kernel

```
#pragma acc kernels
  structured-block
```

```
!$acc kernels
  code-block
!$acc end kernels
```

```c
#pragma acc kernels
{
  // A first kernel generated here
  for (int i = 0; i < n; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
  }
  // A second kernel generated here
  for (int i = 0; i < n; i++) {
    y[i] = i*x[i] + y[i];
    sum += y[i];
  }
}
```

```fortran
!$acc kernels
  ! A first kernel generated here
  do i = 0, n
    x(i) = 1.0
    y(i) = 2.0
  end do
  ! A second kernel generated here
  do i = 0, n
    y(i) = i*x(i) + y(i)
    sum = sum + y[i]
  end do
!$acc end kernels
```

# The parallel construct

The `parallel` construct starts parallel execution on the device

```
#pragma acc parallel
  structured-block
```

```
!$acc parallel
  code-block
!$acc end parallel
```

The `loop` construct identifies a loop eligible for parallelization

```
#pragma acc loop
  for-loop
```

```
!$acc loop
  do-loop
```

```c
#pragma acc parallel
{
  #pragma acc loop
  for (int i = 0; i < n; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
  }

  #pragma acc loop
  for (int i = 0; i < n; i++) {
    y[i] = i*x[i] + y[i];
    sum += y[i];
  }
}
```

```fortran
!$acc parallel
  !$acc loop
  do i = 0, n
    x(i) = 1.0
    y(i) = 2.0
  end do

  !$acc loop
  do i = 0, n
    y(i) = i*x(i) + y(i)
    sum = sum + y[i]
  end do
!$acc end parallel
```

# Data movement clauses

OpenACC data movement directives are very similar to the OpenMP directives: like with the `target` construct you can use data clauses with the `kernels` or parallel `construct`

```
#pragma acc kernels clause(list)              !$acc kernels clause(list)

#pragma acc parallel clause(list)             !$acc parallel clause(list)
```

| Clause | Description |
|---|---|
| create | allocate memory on the device |
| delete | deallocate memory on the device |
| copyin | allocate memory on the device and copy the original values from the host to the device |
| copyout | allocate memory on the device and copy the values from the device to the host |
| copy | combination of copyin and copyout |

# Data region

Structured data region data region are created using the `data` directive

```
#pragma acc data clause(list)
   structured-block
```

```
!$acc data clause(list)
     code-block
!$acc end data
```

For unstructured data regions:

- data movement or allocation to the device is done with the `enter data` directive
- data movement or deallocation from the device are done with the `exit data` directive

```
#pragma acc enter data clause(list)
```

```
!$acc enter data clause(list)
```

```
#pragma acc exit data clause(list)
```

```
!$acc exit data clause(list)
```

# The update directive

- you can update data in the middle of an data region, you can use the `update` directive clauses:

  - `host`: data on the host is updated with data from the device (you can also use `self`)

  - `device`: data on the device is updated with the data from the host

- this directive can be used in the middle of a structured or unstructured data region

```c
#pragma acc data map(tofrom:a[0:n])
{
  // do something with a on the device
  #pragma acc update host(a[0:n])
  // do something with a on the host
  #pragma acc update device(a[0:n])
  // do something with a on the device
}
```

```fortran
!$acc data map(tofrom:a)
  ! do something with a on the device
  !$acc update host(a)
  ! do something with a on the host
  !$acc update device(a)
  ! do something with a on the device
!$acc end data
```

# OpenACC saxpy

We have two options to implement our saxpy example with OpenACC

- using a `kernel` directive and let the compiler analyze our code and generate a kernel that will execute on the GPU

- using a combined `parallel loop` directive to explicitly indicate to the compiler where the source of parallelism is

```
#pragma acc kernels copyin(x[0:n]) copy(y[0:n])
for (int i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}


!$acc kernels copyin(x) copy(y)
do i = 1,n
  y(i) = a * x(i) + y(i)
end do
!$acc end kernels
```

```
#pragma acc parallel loop copyin(x[0:n])
copy(y[0:n])
for (int i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}


!$acc parallel loop copyin(x) copy(y)
do i = 1,n
  y(i) = a * x(i) + y(i)
end do
!$acc end parallel loop
```

# Compilers

OpenACC is not as widely supported by compilers as OpenMP:

- the best implementation is the NVIDIA compiler
- it is possible to use GCC to target both NVIDIA and AMD GPUs
- some commercial compilers, like the HPE Cray one include support for OpenACC

## NVIDIA HPC SDK (NVIDIA only)

```
nvc/nvfortran –acc –Minfo=acc –gpu=<ccXY> <source>
```

## GCC (NVIDIA, *ok performance with recent version*)

```
gcc/gfortran –fopenacc -foffload=nvptx-none <source>
```

# OpenACC Jacobi

- the `parallel loop` construct create a region to be executed on the device and indicate that the outer loop is parallelizable

- the `loop` directive on the inner loop indicates to the compiler that it is parallelizable as well

```
#pragma acc data copy(uold[0:n*m]) create(unew[0:n*m])
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma acc parallel loop reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    #pragma acc loop reduction(max:err)
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                            +  uold[(j-1)*m +     i] + uold[(j+1)*m +     i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
} // end of the data region
```

# OpenACC Jacobi

Similarly to to what we did with the OpenMP version, we can also collapse the nested loop nest with the collapse clause.

```c
#pragma acc data copy(uold[0:n*m]) create(unew[0:n*m])
while (err > tol && iter < iter_max) {
  err = 0.0;

  #pragma acc parallel loop collapse(2) reduction(max:err)
  for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
      unew[j*m + i] = 0.25 * (uold[j*m     + (i+1)] + uold[j*m     + (i-1)]
                            +  uold[(j-1)*m +    i] + uold[(j+1)*m +    i]);

      err = fmax(err, fabs(unew[j*m + i] - uold[j*m + i]));
    }
  }

  // swap values, uold <- unew
} // end of the data region
```

# Coalescent memory access

## Coalescent memory access



memory block
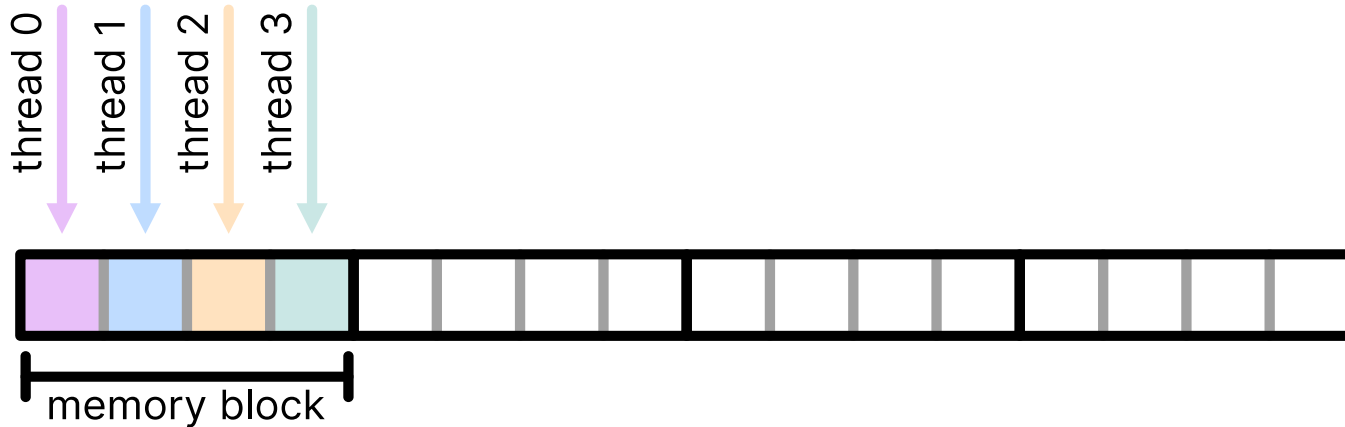
## Uncoalescent memory access
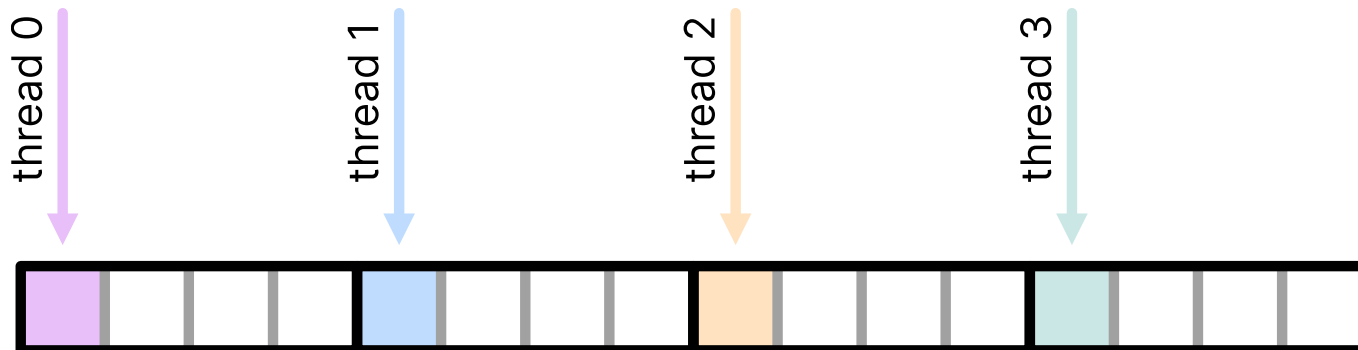


Coalesced memory access refers to combining multiple memory accesses into a single transaction

- when a thread access the GPU global memory it always access a the memory in chunks

- if other threads access the same chunk at the same time then the chunk can be reused

- the most efficient access is when threads read or write contiguous memory locations

- strided memory access is not optimal as more memory transactions are reqired to read/write the same amount of data

# AoS and SoA

**Array of Structures:** cache friendly



```
struct point {
    float x;
    float y;
    float z;
};
struct point points[n];
```

**Structure of arrays:** coalescent access



```
struct points_list {
    float x[n];
    float y[n];
    float z[n];
};
struct points_list points;
```

# Wrapping-up

OpenMP and OpenACC allows you to target GPUs with a few directives added to your code. While adding these directives is relatively easy:

**Transferring data between the host and the device is an expensive process**

- data transfer may be the main bottleneck when running on a accelerator is not handled carefully
- only transfer data required on the device
- try to keep the data on the device as long as possible
- use structured data region (`target data`) or unstructured data region (`target enter/exit data`)

**You need sufficient parallelism in order to achieve good performance**

- need to expose more parallelism that for CPUs
- can be achieved by distributing loops across teams and across threads
- for tightly nested loop collapsing is also an option