First steps in OpenACC

Francesco Andreucci
SISSA

ICTP CIFRA Magurele School,
3/07/2025

How write a code that exploits a GPU

Your application

Libraries (cuFFT…)	Directives (openACC/MP)	Programming languages (CUDA,HIP)
Replace functions Guaranteed perf Fixed interface	Custom code Portable Compiler helps	High flexibility Max perf “Low” level

Effort →

How write a code that exploits a GPU

Your application

https://www.openacc.org/resources

OpenACC

More Science, Less Programming

R
H
Fixed interface

Compiler helps

Low level

Effort

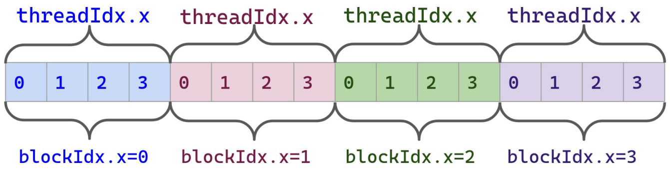
What can you do with OpenACC?

- Runs on different compilers (nvc/pgi, gcc…) and different accelerators (NVIDIA,AMD,multi core CPU…)

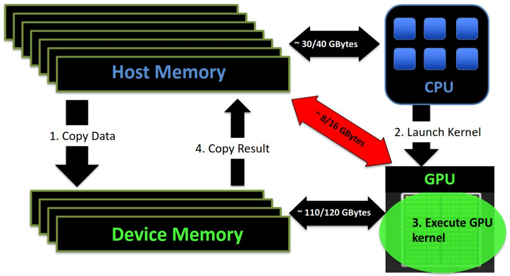
OpenACC
More Science, Less Programming

More (asynch..)

Loop parallelization



Data management



First taste of OpenACC

```
for (int i=0; i<N; i++){
    a[i]=1; b[i]=2;
}
```

```
#pragma acc parallel loop
for (int i=0; i<N; i++){
    a[i]=1; b[i]=2;
}
```

The compiler interprets your directives and **generates GPU CUDA code**

OpenACC is directive-based:

- Directives are **incrementally** added on top of the serial cpu-code **(no refactoring/code duplication)**
- If the compiler does not recognize a directive it **is treated as a comment**
- Supports both C/C++ and Fortran
- Handles both **loop parallelization** and **data management**

First taste of OpenACC

```
do i=1,N
 a(i)=1
end do
```

```
!$acc parallel loop
do i=1,N
    a(i)=1;
end do
```

The compiler interprets your directives and **generates GPU CUDA code**

OpenACC is directive-based:

- Directives are **incrementally** added on top of the serial cpu-code **(no refactoring/code duplication)**
- If the compiler does not recognize a directive it **is treated as a comment**
- Supports both C/C++ and Fortran
- Handles both **loop parallelization** and **data management**

Pragma-directives syntax

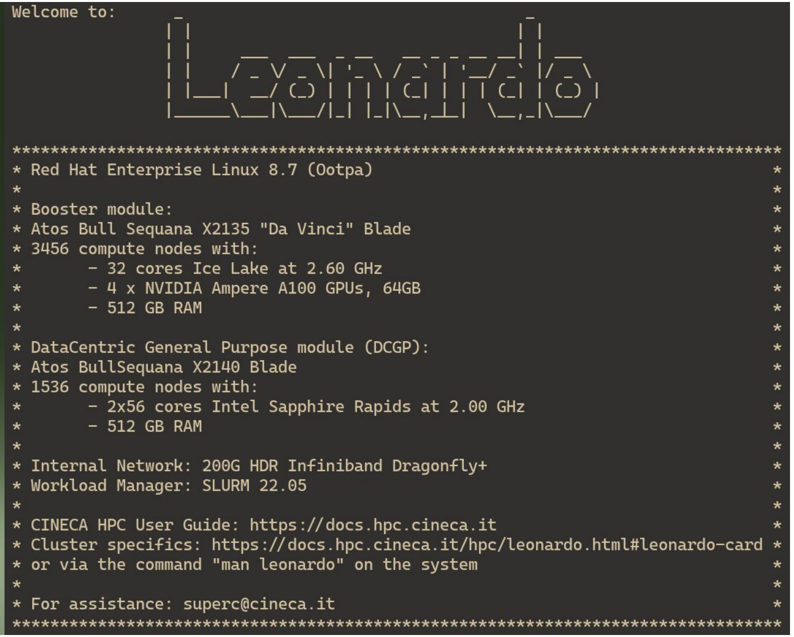
C/C++	`#pragma acc {directives} {clauses}`
Fortran	`!$acc {directives} {clauses}`

- **Directives** are **standalone** commands that tell the compiler to alter the code in some way (i.e memory copies, parallelize loops…)

- **Clauses** are **auxiliary** specifiers that refine the action of the directives

How to compile(on Leonardo, Ampere A100 GPU cc=80):

```
nvc       -o exe myprog.c   -acc -gpu=cc80,cuda12.3 -Minfo=acc
nvc++     -o exe myprog.cpp -acc -gpu=cc80,cuda12.3 -Minfo=acc
nvfortran -o exe myprog.f90 -acc -gpu=cc80,cuda12.3 -Minfo=acc
```



```
Welcome to:

********************************************************************************
* Red Hat Enterprise Linux 8.7 (Ootpa)                                         *
*                                                                              *
* Booster module:                                                              *
* Atos Bull Sequana X2135 "Da Vinci" Blade                                     *
* 3456 compute nodes with:                                                     *
*       - 32 cores Ice Lake at 2.60 GHz                                        *
*       - 4 x NVIDIA Ampere A100 GPUs, 64GB                                    *
*       - 512 GB RAM                                                           *
*                                                                              *
* DataCentric General Purpose module (DCGP):                                   *
* Atos BullSequana X2140 Blade                                                 *
* 1536 compute nodes with:                                                     *
*       - 2x56 cores Intel Sapphire Rapids at 2.00 GHz                         *
*       - 512 GB RAM                                                           *
*                                                                              *
* Internal Network: 200G HDR Infiniband Dragonfly+                             *
* Workload Manager: SLURM 22.05                                                *
*                                                                              *
* CINECA HPC User Guide: https://docs.hpc.cineca.it                            *
* Cluster specifics: https://docs.hpc.cineca.it/hpc/leonardo.html#leonardo-card *
* or via the command "man leonardo" on the system                              *
*                                                                              *
* For assistance: superc@cineca.it                                             *
********************************************************************************
```

Let’s move on the cluster…

-Minfo=all output

```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}

#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}

#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

-Minfo=all output

```
[fandreuc@login02 cifra25]$ nvc -o exe.x openacc1.c    -Minfo=all -acc -gpu=cc80,cuda12.3
main:
     12, Generating implicit firstprivate(i,n)
         Generating NVIDIA GPU code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     12, Generating implicit copyout(b[:100000],a[:100000]) [if not already present]
     18, Generating implicit firstprivate(n,j)
         Generating NVIDIA GPU code
         21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     18, Generating implicit allocate(a[:100001]) [if not already present]
         Generating implicit copyin(a[1:100000]) [if not already present]
         Generating implicit copy(b[:100000]) [if not already present]
         Generating implicit copyout(a[:100000]) [if not already present]
     24, Generating implicit firstprivate(n,k)
         Generating NVIDIA GPU code
         27, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     24, Generating implicit copyin(a[:100000]) [if not already present]
         Generating implicit copyout(c[:100000]) [if not already present]
         Generating implicit copyin(b[:100000]) [if not already present]
```

```
#prag
for(i

}

#prag
for(i

}

#prag
for(i

}
```

-Minfo=all output

```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}

#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}

#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

```
Generating implicit firstprivate(i,N)
Generating NVIDIA GPU code
22, #pragma acc loop gang, vector(128)
    /* blockIdx.x threadIdx.x */
Generating implicit copyout(b[:10000000],a[:10000000])
[if not already present]
```

-Minfo=all output

```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}
```

```
Generating implicit firstprivate(i,N)
Generating NVIDIA GPU code
22, #pragma acc loop gang, vector(128)
    /* blockIdx.x threadIdx.x */
Generating implicit copyout(b[:10000000],a[:10000000])
[if not already present]
```

```
#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}
```

```
21, #pragma acc loop gang, vector(128) /
Generating implicit allocate(a[:100001])
Generating implicit copyin(a[1:100000])
Generating implicit copy(b[:100000]) [if
Generating implicit copyout(a[:100000])
Generating implicit firstprivate(n,k)
Generating NVIDIA GPU code
```

```
#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

-Minfo=all output

```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}
```

```
Generating implicit firstprivate(i,N)
Generating NVIDIA GPU code
22, #pragma acc loop gang, vector(128)
    /* blockIdx.x threadIdx.x */
Generating implicit copyout(b[:10000000],a[:10000000])
[if not already present]
```

```
#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}
```

```
21, #pragma acc loop gang, vector(128) /
Generating implicit allocate(a[:100001])
Generating implicit copyin(a[1:100000])
Generating implicit copy(b[:100000]) [if
Generating implicit copyout(a[:100000])
Generating implicit firstprivate(n,k)
Generating NVIDIA GPU code
```

```
#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

```
Generating implicit firstprivate(j,N)
Generating NVIDIA GPU code
34, #pragma acc loop gang, vector(128)
    /* blockIdx.x threadIdx.x */
Generating implicit copyin(a[:10000000])
Generating implicit copyout(c[:10000000])
Generating implicit copyin(b[:10000000])
```

Memory transfers

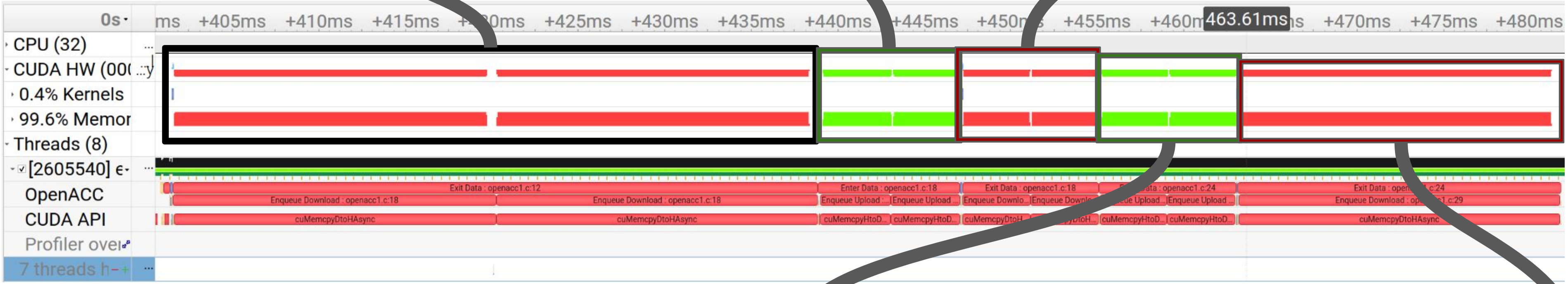
```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}
```

out a,b

in a,b

```
#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}
```

out a,b



in a,b

```
#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

out c

Memory transfers

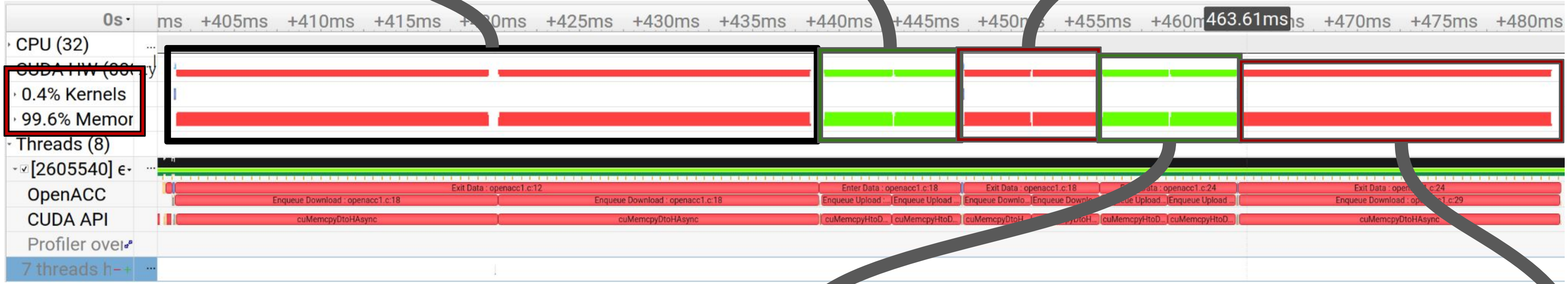
```
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}
```

out a,b

in a,b

```
#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}
```

out a,b



in a,b

```
#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

out c

….99.6% of the GPU activity is spent in memory transfers!

Data Management

```c
#pragma acc parallel loop
for(int i=0; i<n; i++){
        a[i]=3;
        b[i]=4;
}

#pragma acc parallel loop
for(int j=0; j<n; j++){
        a[j] = 2*a[j];
        b[j] = b[j]+1;
}

#pragma acc parallel loop
for(int k=0; k<n; k++){
        c[k] = a[k]+b[k];
}
```

- What we just saw is called **implicit** data management
- This can be **very inefficient** especially if the parallel directive is iteratively invoked inside a loop

Data Management: main points

- Data must be visible to the **device** when running in a **parallel region**

- After a parallel region, we might want to get back the updated data from the device on the host

- To maximize the performance **one should minimize data transfers between host/device**

Data clauses

- Data clauses are used to tell the compiler which data we want to move

```
#pragma acc parallel loop copy(a[0:n])
for(i=0;i<n;i++){
        a[i] = a[i] +1;
}
```

a[start_index:length]	C/++
a(start_index:end_index)	F90

- You can also **slice the array** and copy only what you need

```
#pragma acc parallel loop copy(a[5:m])
for(i=5;i<m;i++){
        a[i] = a[i] +1;
}
```

Data clauses

- Data clauses are used to tell the compiler which data we want to move

```
!$acc parallel loop copy(a(1:n))
do i=1,n
        a(i) = a(i) +1
end do
```

`a[start_index:length]`	C/++
`a(start_index:end_index)`	F90

- You can also **slice the array** and copy only what you need

```
!$acc parallel loop copy(a(5:m))
do i=5,m
        a(i) = a(i) +1
end do
```

Data clauses

• copy(list)	• Allocates memory on GPU, copies data from host when entering the region and copies data back from device at the end
• copyin(list)	• Allocates memory on GPU, copies data from host when entering the region
• copyout(list)	• Allocates memory on GPU, copies data back from device at the end
• create(list)	• Allocates memory on GPU (useful for tmp buffers)
• delete(list)	• Frees memory on the GPU(!)

Structured data regions

```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    a[i] = 3; b[i]=4;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    a[i] = 2*a[i]; b[i] = b[i]+1
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
 }
```

a copied in and out

a,b copied in and out

a,b copied in and c copied out

Structured data regions

```
#pragma acc data copyin(a[0:N],b[0:N])
copyout(c[0:N])
{
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 3; b[i]=4;
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 2*a[i]; b[i] = b[i]+1
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     c[i] = a[i] + b[i];
  }

}
```

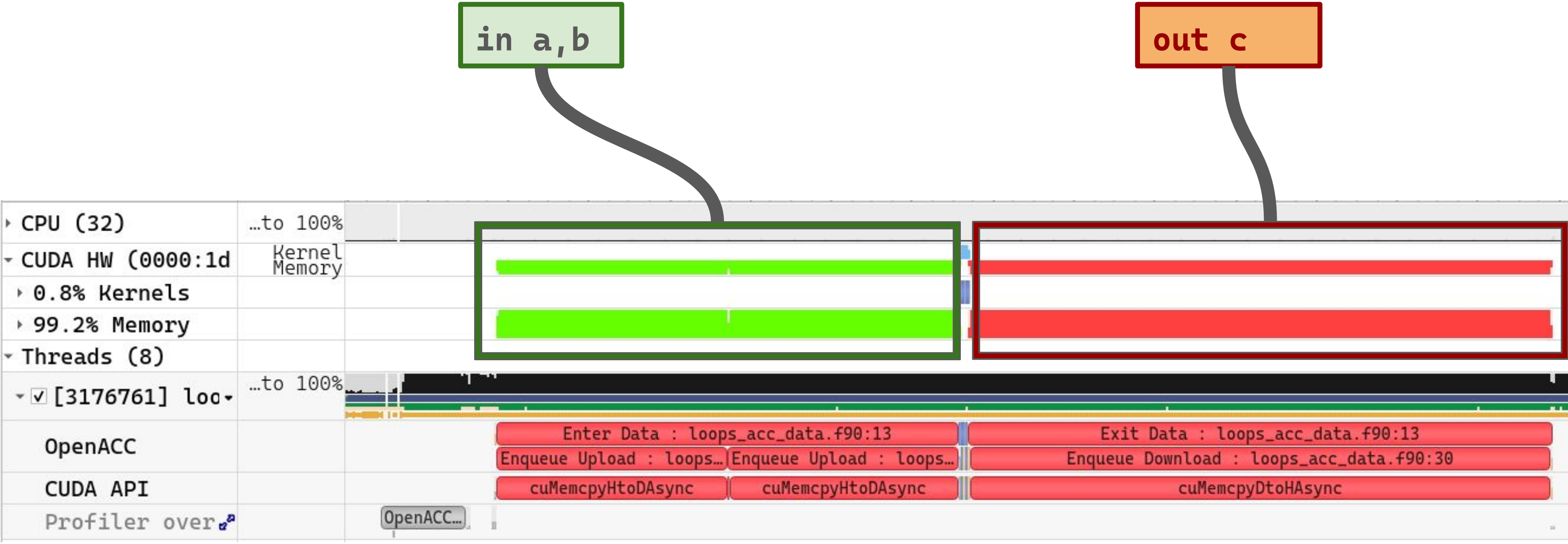
- Inside the **data region** the runtime knows a,b,c are in the GPU **and does not copy them back and forth**
- As we saw before, a lone **parallel(/kernels)** directive open an **implicit data region** that spans the parallel region

Structured data regions

```
#pragma acc data copyin(a[0:N],b[0:N])
copyout(c[0:N])
{
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 3; b[i]=4;
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 2*a[i]; b[i] = b[i]+1
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     c[i] = a[i] + b[i];
  }

}
```

```
main:
     15, Generating copyin(a[:n],b[:n])
         [if not already present]
         Generating copyout(c[:n])
         [if not already present]
         Generating implicit firstprivate(n,i)
         Generating NVIDIA GPU code
         17, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
     20, Generating implicit firstprivate(n,j)
         Generating NVIDIA GPU code
         31, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
     34, Generating implicit firstprivate(n,k)
         Generating NVIDIA GPU code
         37, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
```

Structured data regions



Unstructured data regions

```
#pragma acc enter data \ copyin(a[0:N],b[0:N])
create(c[0:N])

 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 3; b[i]=4;
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     a[i] = 2*a[i]; b[i] = b[i]+1
  }
 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
     c[i] = a[i] + b[i];
  }

#pragma acc exit data delete(a,b) \copyout(c)
```

- **enter data**
  + copyin, create
- **exit data**
  + copyout, delete

(Un)structured data regions

Unstructured data

- Multiple start/end points
- Can branch across scopes
- Memory must be explicitly deallocated

```
#pragma acc enter data \
  copyin(a[0:n],b[0:n]) create(c[0:n])

#pragma acc parallel loop
     for(i=0;i<n;i++) c[i]=a[i]+b[i];

#pragma acc exit data copyout(c[0:n])
delete(a,b)
```

Structured data

- Explicit start/end points
- Within a single scope
- Memory is deallocated after the data region

```
#pragma acc data copyin(a[0:n],b[0:n])\
  copyout(c[0:n])
{
#pragma acc parallel loop
     for(i=0;i<n;i++) c[i]=a[i]+b[i];
}
```

Unstructured data regions

```
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int));

        return ptr;
 }

void deallocate(int * ptr)
 {
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);

        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Unstructured data regions

```c
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int));

        return ptr;
 }
void deallocate(int * ptr)
 {
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);

        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Data allocation

Unstructured data regions

```
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int));

        return ptr;
 }

void deallocate(int * ptr)
 {
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);

        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Data allocation

Data utilization

Unstructured data regions

```c
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int));

        return ptr;
 }

void deallocate(int * ptr)
 {
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);

        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Data allocation

Data utilization

Data deallocation

Unstructured data regions

```
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int)); );
        #pragma acc enter data create(c[0,size])
        return ptr;
 }

void deallocate(int * ptr)
 {
        #pragma acc exit data delete(ptr)
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);
        #pragma acc parallel loop
        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Data allocation

Data utilization

Data deallocation

Unstructured data regions

```
int * allocate(int size)
 {

        int * ptr = (int *) malloc(size*sizeof(int)); );
        #pragma acc enter data create(c[0,size])
        return ptr;
 }

void deallocate(int * ptr)
 {
        #pragma acc exit data delete(ptr)
        free(ptr);
 }

int main()
 {
        int *ptr = allocate(100);
        #pragma acc parallel loop
        for(int i=0; i<100; i++){
            ptr[i]=0;
        }
        deallocate(ptr)

        return 0;
 }
```

Data allocation

Data deallocation

Data utilization

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
        #pragma acc parallel loop
         for(int i=0;i<n;i++){
            a[i]=3; b[i]=4;
         }

         printf(a); printf(b);

         for(int j=0;i<n;j++){
            a[j]=2*a[j]; b[j]=b[j]+1;
         }
         #pragma acc parallel loop
         for(int k=0;k<n;k++){
            c[k] = a[k] + b[k];
         }
}
         printf(c);
```

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
        #pragma acc parallel loop
         for(int i=0;i<n;i++){
            a[i]=3; b[i]=4;
         }

         printf(a); printf(b);

         for(int j=0;i<n;j++){
            a[j]=2*a[j]; b[j]=b[j]+1;
         }
         #pragma acc parallel loop
         for(int k=0;k<n;k++){
            c[k] = a[k] + b[k];
         }
}
         printf(c);
```

a and b are updated **on device**

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
    #pragma acc parallel loop
    for(int i=0;i<n;i++){
        a[i]=3; b[i]=4;
    }

    printf(a); printf(b);

    for(int j=0;i<n;j++){
        a[j]=2*a[j]; b[j]=b[j]+1;
    }
    #pragma acc parallel loop
    for(int k=0;k<n;k++){
        c[k] = a[k] + b[k];
    }
}
    printf(c);
```

a and b are updated **on device**

a and b are **printed from host (un updated)**

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
         #pragma acc parallel loop
          for(int i=0;i<n;i++){
             a[i]=3; b[i]=4;
          }

          printf(a); printf(b);

          for(int j=0;i<n;j++){
             a[j]=2*a[j]; b[j]=b[j]+1;
          }
          #pragma acc parallel loop
          for(int k=0;k<n;k++){
             c[k] = a[k] + b[k];
          }
}
          printf(c);
```

a and b are updated **on device**

a and b are **printed from host (un updated)**

a and b are **updated on host**

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
    #pragma acc parallel loop
     for(int i=0;i<n;i++){
        a[i]=3; b[i]=4;
     }

     printf(a); printf(b);

     for(int j=0;i<n;j++){
        a[j]=2*a[j]; b[j]=b[j]+1;
     }
     #pragma acc parallel loop
     for(int k=0;k<n;k++){
        c[k] = a[k] + b[k];
     }
}
     printf(c);
```

a and b are updated **on device**

a and b are **printed from host (un updated)**

a and b are **updated on host**

c is **updated on device**
c is **copied out to host**

Keeping host and device in synch

```
#pragma acc data copyin(a[0:n],b[0:n]) \
                 copyout(c[0:n])
{
   #pragma acc parallel loop
    for(int i=0;i<n;i++){
       a[i]=3; b[i]=4;
    }

    printf(a); printf(b);

    for(int j=0;i<n;j++){
       a[j]=2*a[j]; b[j]=b[j]+1;
    }
    #pragma acc parallel loop
    for(int k=0;k<n;k++){
       c[k] = a[k] + b[k];
    }
}
    printf(c);
```

a and b are updated **on device**

a and b are **printed from host (un updated)**

a and b are **updated on host**

c is **updated on device**
c is **copied out to host**

c is **printed from host**

Keeping host and device in synch

If you try to copy data already PRESENT on the GPU, the copy is not done.

```
< a[] modified on the host >

#pragma acc enter data copyin(a[0:N]) → host and device copies are in sync

< a modified on the host > → host and device copies are out of sync

#pragma acc data copyin(a[0:N]) ! copy is ignored

< a used on the GPU > → host and device copies are out of sync
```

Keeping host and device in synch

If you try to copy data already PRESENT on the GPU, the copy is not done.

```
<a[] is changed on device>

#pragma acc update host(a[0:n])

<a[] is changed on host>

#pragma acc update device(a[0:n])
```

a is copied from GPU to CPU

a is copied from CPU to GPU

- You can also shape the update: **update host(a[start_index:count])**
- You can also use: **update self = update host**

Keeping host and device in synch

If you try to copy data already PRESENT on the GPU, the copy is not done.

```
<a() is changed on device>
!$acc update host(a(1:n))
<a() is changed on host>
!$acc update device(a(1:n))
```

a is copied from GPU to CPU

a is copied from CPU to GPU

- You can also shape the update: `update host(a(start_index:end_index))`
- You can also use: `update self = update host`

```
Welcome to:

*****************************************************************************************
* Red Hat Enterprise Linux 8.7 (Ootpa)                                                  *
*                                                                                       *
* Booster module:                                                                       *
* Atos Bull Sequana X2135 "Da Vinci" Blade                                              *
* 3456 compute nodes with:                                                              *
*       - 32 cores Ice Lake at 2.60 GHz                                                 *
*       - 4 x NVIDIA Ampere A100 GPUs, 64GB                                             *
*       - 512 GB RAM                                                                    *
*                                                                                       *
* DataCentric General Purpose module (DCGP):                                            *
* Atos BullSequana X2140 Blade                                                          *
* 1536 compute nodes with:                                                              *
*       - 2x56 cores Intel Sapphire Rapids at 2.00 GHz                                  *
*       - 512 GB RAM                                                                    *
*                                                                                       *
* Internal Network: 200G HDR Infiniband Dragonfly+                                      *
* Workload Manager: SLURM 22.05                                                         *
*                                                                                       *
* CINECA HPC User Guide: https://docs.hpc.cineca.it                                     *
* Cluster specifics: https://docs.hpc.cineca.it/hpc/leonardo.html#leonardo-card *
* or via the command "man leonardo" on the system                                       *
*                                                                                       *
* For assistance: superc@cineca.it                                                      *
*****************************************************************************************
```

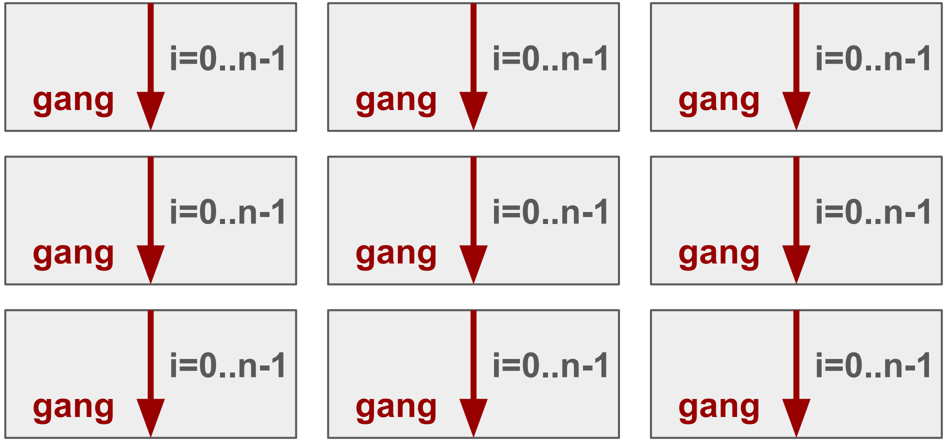
Let’s move on the cluster…

The parallel directive

```
#pragma acc parallel
{
   for(i=0;i<n;i++){
       <code>
   }
}
```

The **parallel** directive:

- Creates a region where the code is run on the GPU **sequentially** (single thread)
- A **gang** is an **independent group of threads** (more details later…)

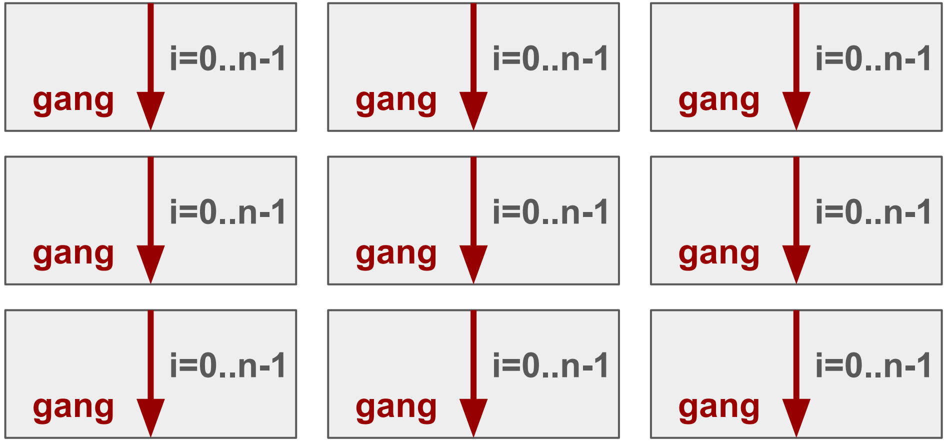


The parallel directive

```
#pragma acc parallel
{
   for(i=0;i<n;i++){
       <code>
   }
}
```

The **parallel** directive:

- Creates a region where the code is run on the GPU **sequentially** (single thread)
- A **gang** is an **independent group of threads** (more details later…)



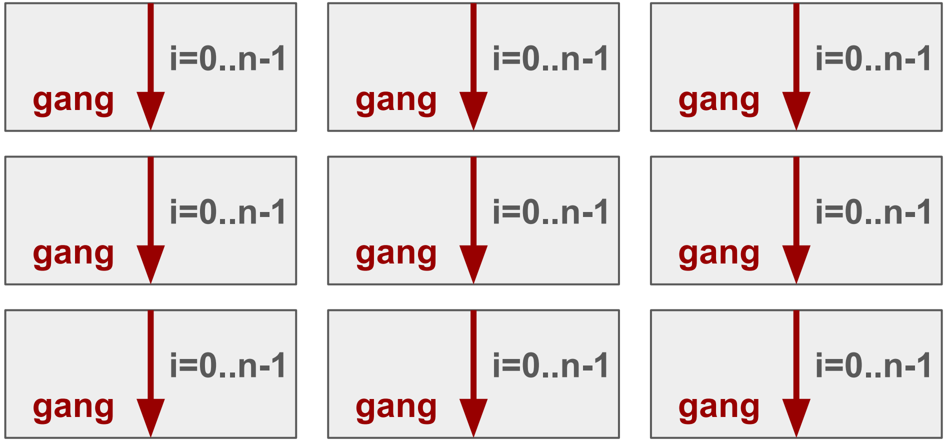
This is ***not*** what we want: the code is run **redundantly** we are wasting GPU resources

The parallel directive

```
!$acc parallel

   do i=1,N
       <code>
   end do
!$acc end parallel
```

The **parallel** directive:

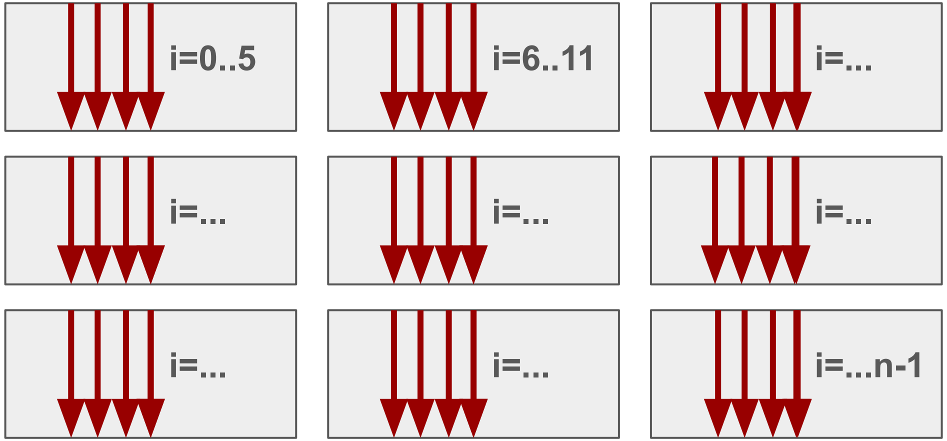
- Creates a region where the code is run on the GPU **sequentially** (single thread)
- A **gang** is an **independent group of threads** (more details later…)



This is ***not*** what we want: the code is run **redundantly** we are wasting GPU resources

The loop directive

```
#pragma acc parallel
{
        #pragma acc loop
        for(i=0;i<n;i++){ <code> }

}
```



The **loop** directive:

- Instructs the compiler to distribute the loop iterations among the gangs

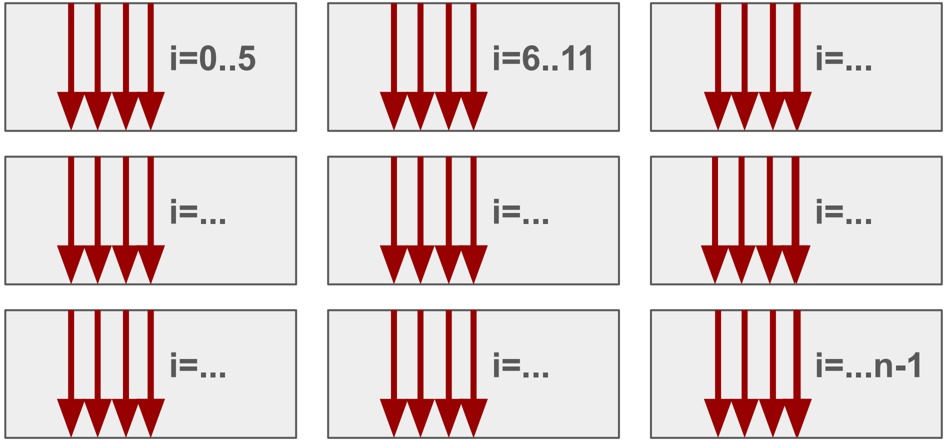
This *is* what we want: each gang (block) takes care of a section of the loop!

The loop directive

```
!$acc parallel
   !$acc loop
   do i=1,N
       <code>
   end do
!$acc end parallel
```

The **loop** directive:

- Instructs the compiler to distribute the loop iterations among the gangs



This *is* what we want: each gang (block) takes care of a section of the loop!

The parallel loop directive

```
#pragma acc parallel loop
for(i=0;i<n;i++){
    <code>
}
```

- The **parallel** and **loop** directives are often combined in the **parallel loop** directive

- The compiler might decide to insert **loop** directives to avoid race conditions!

Quick recap:

- **Parallel** marks the region of the code that will be translated into a GPU kernel (i.e: parallel execution)

- **Loop** is used to tell the compiler to parallelize the **next** loop it finds across the gangs

The parallel loop directive

```
!$acc parallel loop
do i=1,N
       <code>
end do
```

- The **parallel** and **loop** directives are often combined in the **parallel loop** directive

- The compiler might decide to insert **loop** directives to avoid race conditions!

Quick recap:

- **Parallel** marks the region of the code that will be translated into a GPU kernel (i.e: parallel execution)

- **Loop** is used to tell the compiler to parallelize the **next** loop it finds across the gangs

The kernels directive

```
#pragma acc kernels
{
  for(i = 0; i < n; i++)
  a[i] = 0;

  for(j = 0; j < m-1; j++)
  b[j] = b[j+1]-b[j];
}
```

- The **kernels** directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe to do so
- The kernels directive can be applied to regions containing multiple nested loops

The kernels directive

```
#pragma acc kernels
{
  for(i = 0; i < n; i++)
  a[i] = 0;

  for(j = 0; j < n-1; j++)
  b[j] = b[j+1]-b[j];
}
```

C/C++ allows aliasing, so you need to tell the compiler that pointers are not aliased! (using restrict/__restrict__)

- The **kernels** directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe to do so
- The kernels directive can be applied to regions containing multiple nested loops

The kernels directive

```fortran
!$acc kernels
a(:) = 1
b(:) = 2
c(:) = a(:) + b(:)
!$acc end kernels
```

In Fortran aliasing is forbidden by the standard

- The **kernels** directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe to do so
- The kernels directive can be applied to regions containing multiple nested loops
- Compatible with Fortran array notation

Nested loops

```
#pragma acc parallel loop
for(i=0;i<n;i++){
    #pragma acc loop
    for(j=0;j<m;j++){
        a[i*n+j]=0;
    }
}
```

```
#pragma acc kernels
{
    for(i=0;i<n;i++){
        for(j=0;j<m;j++){
            a[i*n+j]=0;
        }
    }
}
```

- The **parallel loop / kernels** directives can be nested to parallelize multi-dimensional loops
- If resources are available, the compiler can exploit more level of parallelism

Nested loops

```
!$ acc parallel loop
do i=1,n
    !$ acc loop
    do j=1,m
        a(i,j)=0
    end do
end do
```

```
!$acc kernels
do i=1,n
    do j=1,m
        a(i,j)=0
    end do
end do
!$acc end kernels
```

- The **parallel loop / kernels** directives can be nested to parallelize multi-dimensional loops
- If resources are available, the compiler can exploit more level of parallelism

Nested loops: collapse clause

```
#pragma acc parallel loop collapse(2)
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        a[i*n+j]=0;
    }
}
```

```
for(ij=0;ij<n*n;ij++){
        int i = ij / n;
        int j = ij % n;
        a[i*n+j]=0;
    }
}
```

- The **collapse(n)** clause collapses the next n **tightly** nested loop in a single one
- Useful for creating larger loops in order to **increase memory locality** and **exploit more parallelism**

Privatizations

```
double tmp[3];

#pragma acc parallel loop private(tmp[0:3])
for( i = 0; i < size; i++ ){
  tmp[0] = i;
  tmp[1] = i+1;
  tmp[2] = i+2;
}
```

(first)private clause allows each thread to have private copies of variables:

- **private** variables are uninitialized.
- **firstprivate** private values are initialized to the last value used on the host.

Privatizations

```
double tmp[3];
tmp[0]=1; tmp[1]=2; tmp[3]=3;
#pragma acc parallel loop private(tmp[0:3])
for( i = 0; i < n; i++ ) {
  tmp[0] = i;
  tmp[1] = i+1;
  tmp[2] = i+2;
  #pragma acc loop
  for ( j = 0; j < n; j++) {
      M[n*i+j] = tmp[0]+tmp[1]+tmp[2]+j;
  }
}
printf(tmp);
-----------------------------------------
OUTPUT:
(1,2,3)
```

The tmp[] array is private in each iteration of the outer loop

This private copy is shared between the threads in the inner loop (same scope)

The value on the host won't be updated!

Privatizations (scalars)

- Scalars are by default private in kernel regions and firstprivate in parallel regions
- You typically don’t have to touch scalars except (e.g.):

  1.) Global vars in C/C++, module vars in Fortran

  2.) Scalar used as an rvalue after the parallel region

The reduction clause

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    dot += a[i] * b[i];
}
```

The reduction clause

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    dot += a[i] * b[i];
}
```

Scalars are firstprivate: the value of dot is not changed on the host!

The reduction clause

```
#pragma acc parallel loop reduction(+:dot)
for (int i = 0; i < N; i++) {
     dot += a[i] * b[i];
}
```

- Each thread will have its own **private copy** and will perform a **partial reduction**

The reduction clause

```
#pragma acc parallel loop reduction(+:dot)
for (int i = 0; i < N; i++) {
    dot += a[i] * b[i];
}
```

- Each thread will have its own **private copy** and will perform a **partial reduction**

- At the end, the local sums are combined safely

The compiler is very keen on inserting reduction clauses where needed, e.g. in this case it would have done so automatically.

The reduction clause

```
#pragma acc parallel loop reduction(+:v[0])
for (int i = 0; i < N; i++) {
     v[0] += a[i] * b[i];
}
```

```
double tmp = v[0];
#pragma acc parallel loop reduction(+:tmp)
for (int i = 0; i < N; i++) {
     tmp += a[i] * b[i];
}
v[0] = tmp;
```

You cannot reduce on

- Array entries
- C/C++ class/struct members
- Fortran derived types members

The reduction clause

```
#pragma acc parallel loop reduction(+:mesh.data[0])
for (int i = 0; i < N; i++) {
     mesh.data[0] += a[i] * b[i];
}
```

```
auto tmp = mesh.data[0];
#pragma acc parallel loop reduction(+:tmp)
for (int i = 0; i < N; i++) {
     tmp += a[i] * b[i];
}
mesh.data[0] = tmp;
```

You cannot reduce on

- Array entries
- C/C++ class/struct members
- Fortran derived types members

The reduction clause

```
!$acc parallel loop reduction(+:mytype%member)
do i = 1, N
     mytype%member = mytype%member + 7 + a(i)
end do
```

```
tmp = mytype%member
!$acc parallel loop reduction(+:tmp)
do i = 1, N
     tmp = tmp + 7 + a(i)
end do
mytype%member = tmp
```

You cannot reduce on

- Array entries
- C/C++ class/struct members
- Fortran derived types members

The reduction clause

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum value	reduction(min:minimum)
&	Bitwise and	reduction(&:val)
|	Bitwise or	reduction(|:val)
&&	Logical and	reduction(&&:val)
||	Logical or	reduction(||:val)

Seq and atomic

```
#pragma acc parallel loop seq
for (int i = 0; i < N; i++) {
    <code>
}
```

The **seq** clause will tell the compiler to run the loop sequentially

!The compiler may insert **seq** clause (**check Minfo**)

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    if(a[i] > thr){
#pragma acc atomic
    n_thr++
    }
}
```

The **atomic** directive will tell the compiler that the threads have to execute the instruction **sequentially**

With **atomic** all threads update the same memory location, no partial accumulation as in **reduction**

Seq and atomic

```
#pragma acc parallel loop seq
for (int i = 0; i < N; i++) {
     <code>
}
```

The **seq** clause will tell the compiler to run the loop sequentially

!The compiler may insert **seq** clause (**check Minfo**)

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    if(a[i] > thr){
#pragma acc atomic update
     n_thr++
    }
}
```

atomic supports several clauses:

- **read/write** — **value=x**
- **update (default)** — **x= expr**
- **capture** — **x= expr; value=x**

Atomic example

```c
//count[i] are some integers

#pragma acc parallel loop
for(int i=0;i<range;i++)
   hist[i]=0;


#pragma acc parallel loop
for(int i=0;i<d;i++) {
#pragma acc atomic
   hist[count[i]]+=1;
}
```

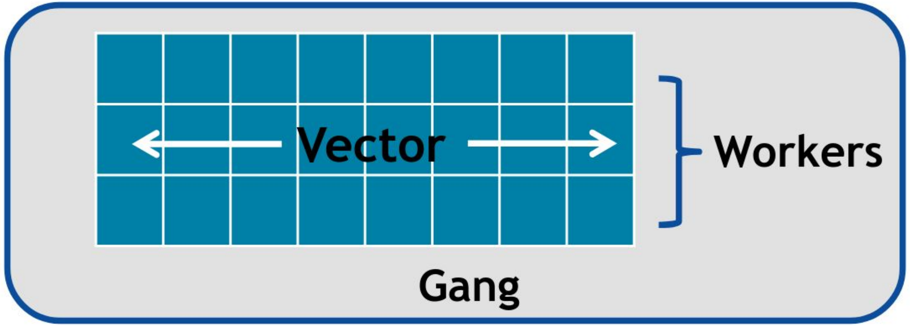
```fortran
//count(i) are some integers

!$acc parallel loop
do i=1,range
    hist(i)=0
end do

!$acc parallel loop
do i=1,d
!$acc atomic
   hist(count(i))+=1;
end do
```

Layers of parallelism

OpenACC expresses 3 layers of parallelism: **gangs,workers,vectors**

- **Vector**: the finest grain
- **Gang**: the coarsest level
- **Worker**: intermediate level
- The correspondence between the openACC expressed parallelism and the hardware parallelism depends on the hardware (e.g. on Intel multicore: a gang is a thread and the vector is an AVX vector instruction)

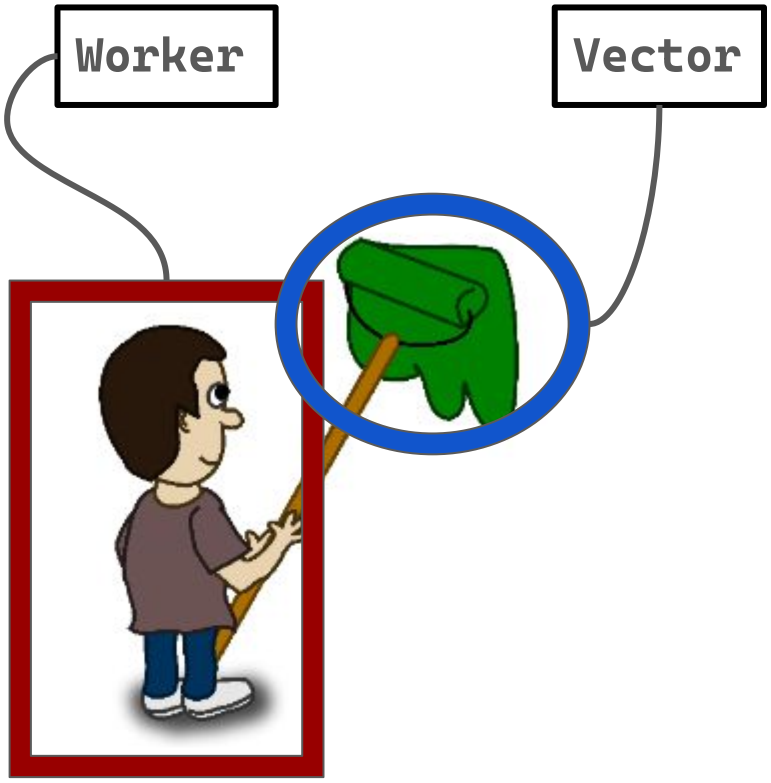
Layers of parallelism



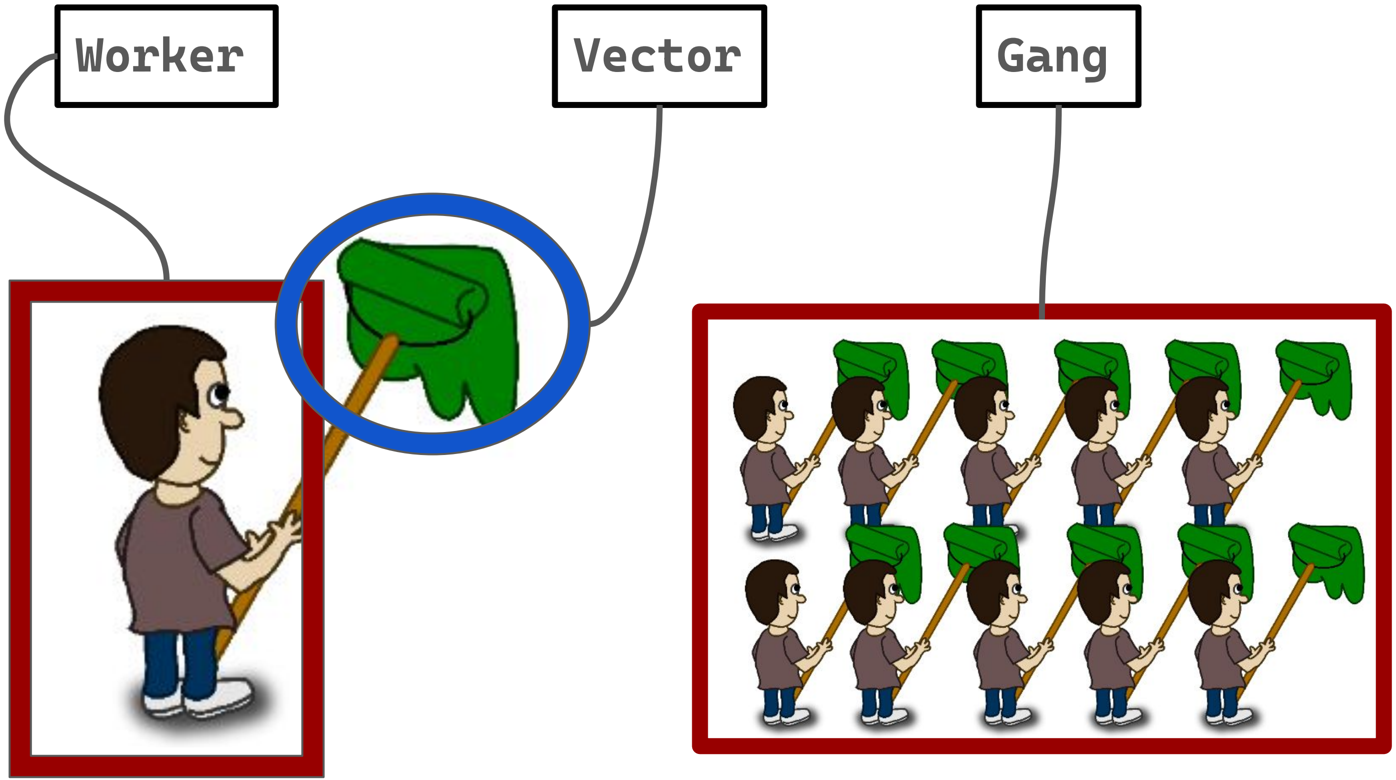
- **Vector** : ThreadIdx.x
- **Worker** : ThreadIdx.y
- **Gang** : Block

- The size of a gang is (**num_workers** * **vector_lenght**)
- **vector_lenght** must be multiple of 32 (warp size)
- These parameters can be specified in the code as clauses

Layers of parallelism

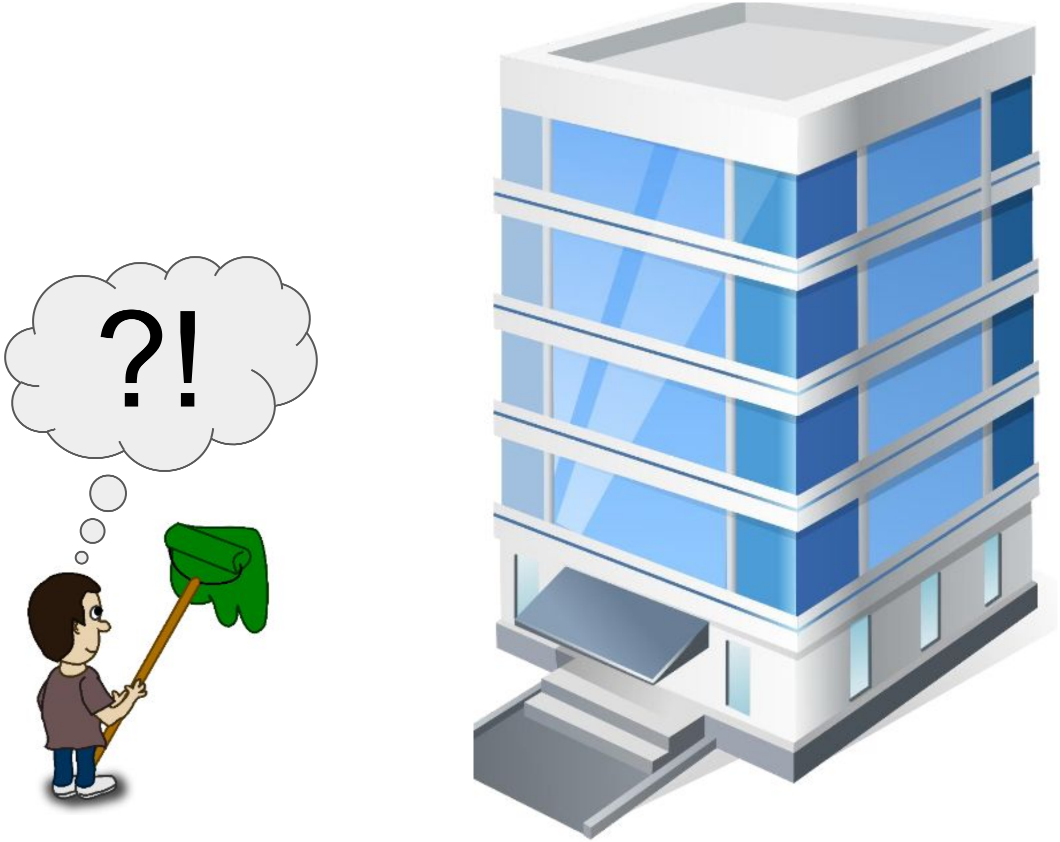


Layers of parallelism



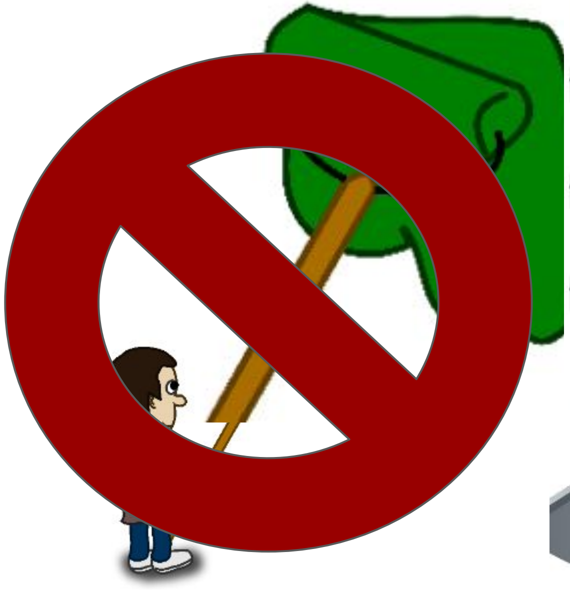
Layers of parallelism

Layers of parallelism



Layers of parallelism

Layers of parallelism



Layers of parallelism

- Independent groups of workers (**gangs**) can clean different floors at the same time

Specify the parallelism

parallel:

- num_gangs(#)
- num_workers(#)
- vector_length(#)

kernel:

- gang(#)
- worker(#)
- vector(#)

```
#pragma acc parallel loop gang
for( i = 0; i < N; i++ )
   #pragma acc loop vector
   for( j = 0; j < M; j++ )
      < loop code >
```

- Typically the compiler generates G gangs,one worker and a vector size of 128

Specify the parallelism

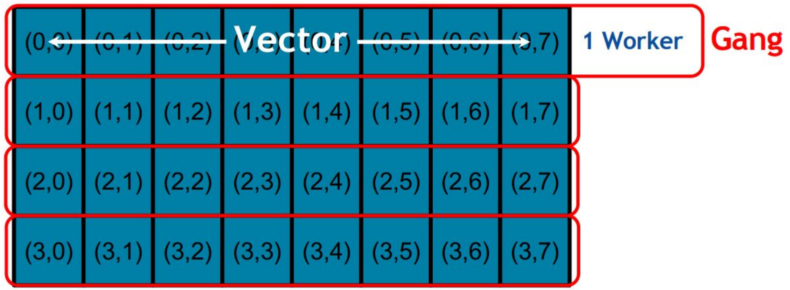
parallel:

- num_gangs(#)
- num_workers(#)
- vector_length(#)

kernel:

- gang(#)
- worker(#)
- vector(#)

```
#pragma acc parallel loop gang
for( i = 0; i < N; i++ )
    #pragma acc loop vector
    for( j = 0; j < M; j++ )
        < loop code >
```



Specify the parallelism

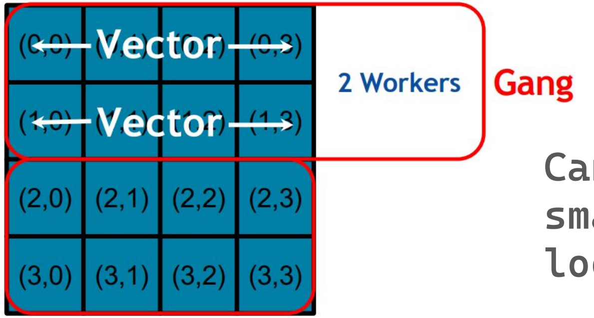
parallel:

- num_gangs(#)
- num_workers(#)
- vector_length(#)

kernel:

- gang(#)
- worker(#)
- vector(#)

```
#pragma acc parallel num_workers(2)
#pragma acc loop gang worker
for( i = 0; i < N; i++ )
    #pragma acc loop vector
    for( j = 0; j < M; j++ )
        < loop code >
```



Can be useful for smaller inner loops

Routine directive

```
#pragma acc routine worker
void square_array(float *arr, int length) {
    #pragma acc parallel loop worker
    for(int i = 0; i < length; ++i) {
        arr[i] *= arr[i];
    }
}

int main(){
    <code>
    #pragma acc parallel loop gang
    for(int i = 0; i < size; ++i){
        square_array(&data[i], 1);
    }
```

- The **routine** directive instructs the compiler to create a device version of the function/subroutine
- As clauses, one can pass the level of parallelism for the loops in the function: **gangs, worker, vector, seq**
- **Other clauses are specified in the API guide**

Routine directive

```
module accroutine
    contains
     real function sqab(a)
        !$acc routine seq
        real :: a
        sqab = sqrt(abs(a))
     end function
end module

subroutine test( x, n )
       use accroutine
       implicit none
       real, dimension(*) :: x
       integer :: n,i
       !$acc parallel loop
       do i = 1, n
          x(i) = sqab(x(i))
       enddo
end subroutine
```

- The **routine** directive instructs the compiler to create a device version of the function/subroutine

- As clauses, one can pass the level of parallelism for the loops in the function: **gangs, worker, vector, seq**

- **Other clauses are specified in the API guide**

- You can also use the accelerated routine inside another function

THANK YOU FOR YOUR ATTENTION!

Exercises

1. Run the examples and play around with the pragmas
2. Implement pi estimation with openACC
3. Implement matrix multiplication using openACC pragmas (try different ways)
4. (Advanced weekend exercise ^_^) Implement mat mul with openACC using different configurations of gangs,worker,vector and compare performance to cublas