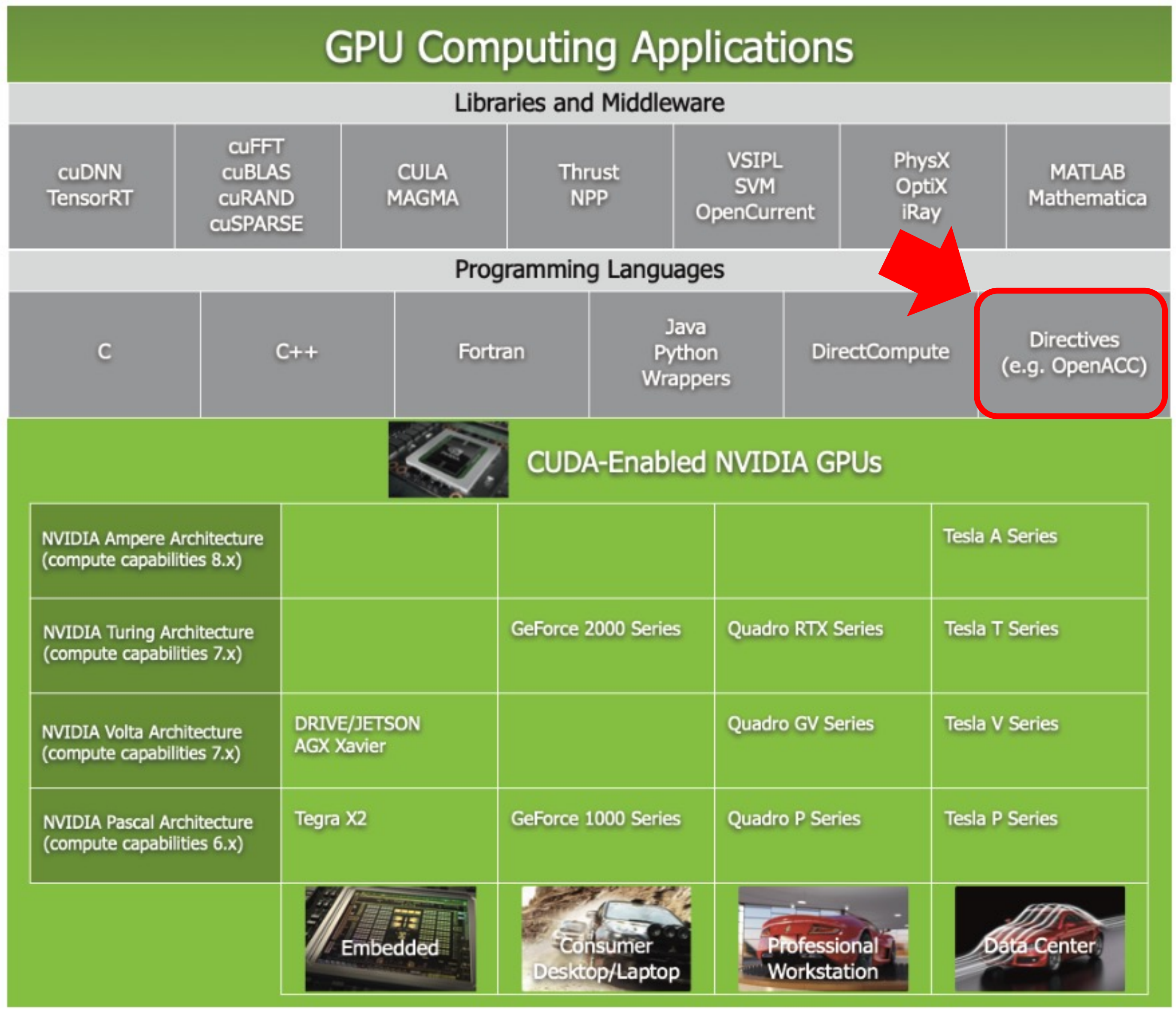
GPUs and Heterogeneous Systems
(programming models and architectures)

OpenACC

CUDA software platform



What OpenACC is

- OpenACC is a specification of compiler directives and API routines for writing parallel code in C/C++ or Fortran

- How it works:
  1. The programmer annotates the existing sequential code to highlight the sections that can be parallelized
  2. The compiler generates the accelerated kernels based on annotations

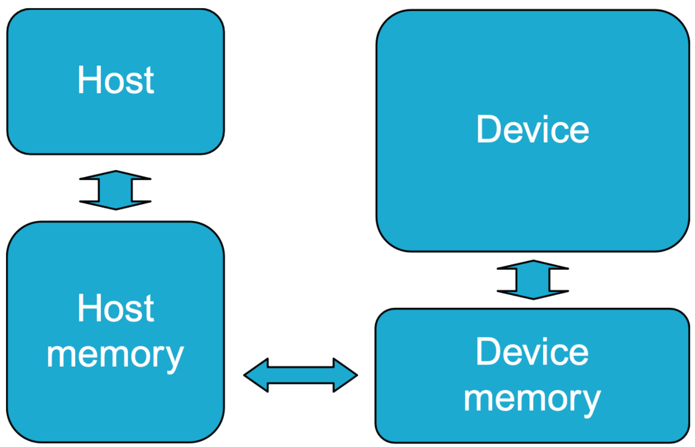
- OpenACC is portable among multiple acceleration platforms
  – Not only NVIDIA GPUs but also other vendors and devices (e.g., multicore)

OpenACC main characteristics

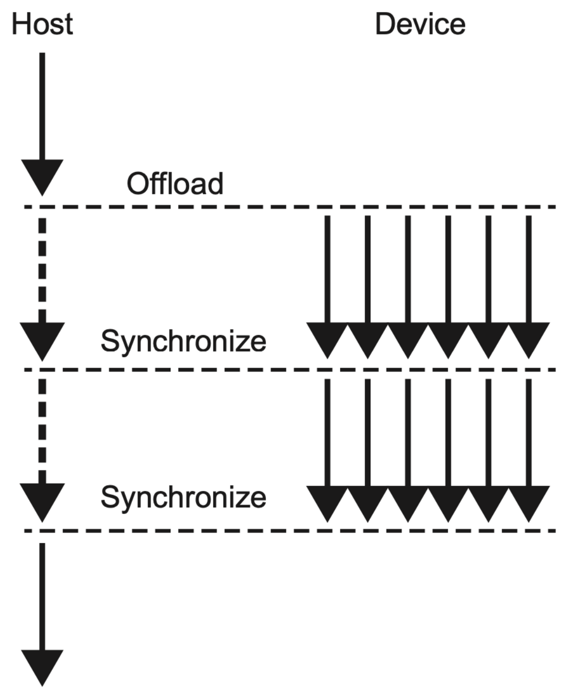
- **High portability**
  - The source code can be targeted to different accelerators by recompiling
- **High programmability**
  - The original source code is simply annotated with directives
- **Lower performance than CUDA**
  - Due to the support of multiple devices and the high level of abstraction specific optimizations cannot be applied by means of annotations

OpenACC architecture and execution models

- Architecture and execution models are similar to CUDA ones

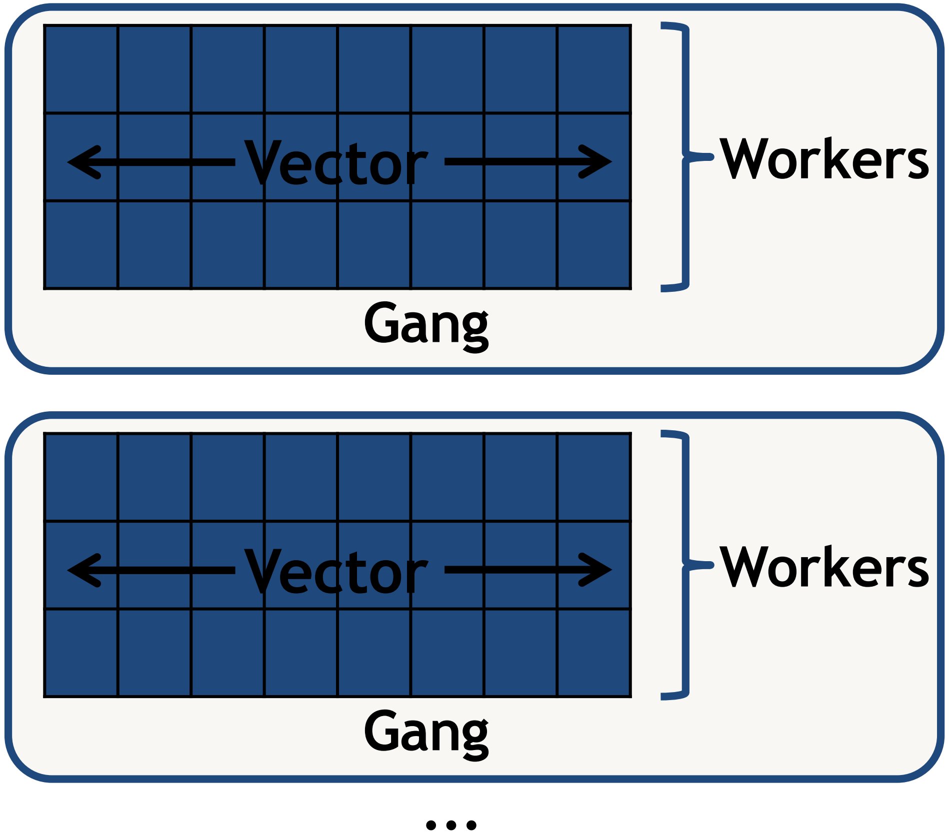


- Device is not necessarily a GPU
- Device and host may be the same physical device



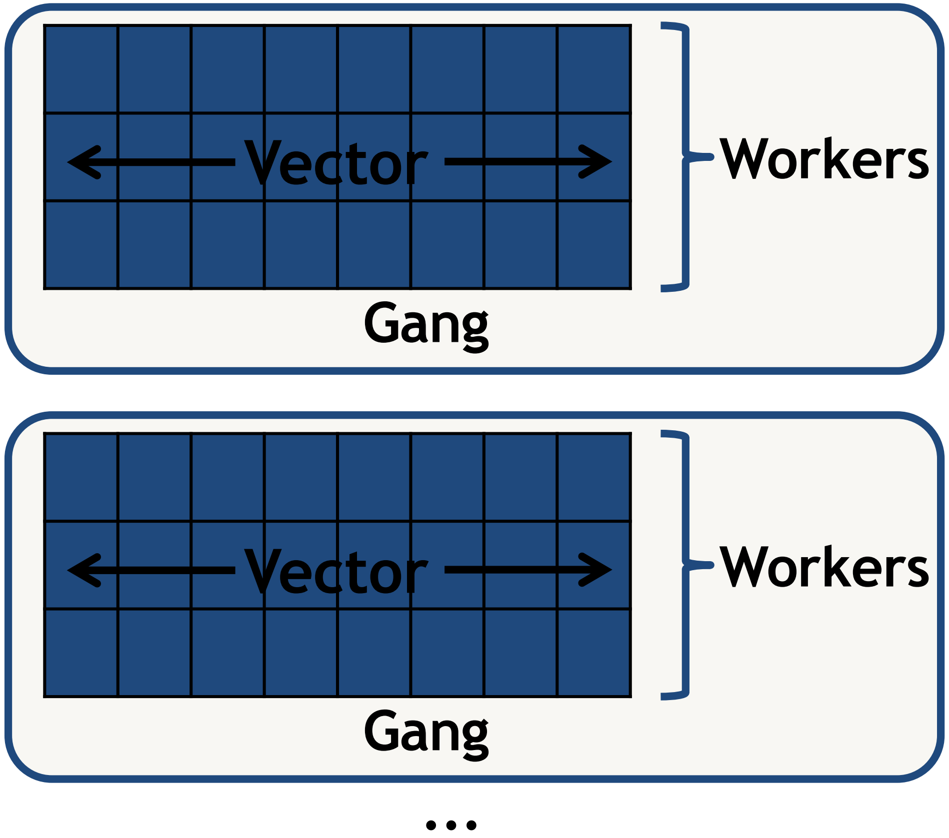
Parallelism model

- 3 levels of parallelism:
  - **Gangs**
    - Fully independent execution units
    - Each one is a group of one or more workers sharing resources
  - **Workers**
    - Each one elaborates a vector of work
    - Workers within the same gang can synchronize
  - **Vectors**
    - The vector executes the same instruction on multiple data (possibly SIMD execution)
    - The vector element is the single processing unit
    - The vector width is the number of elements in the vector



Parallelism model – OpenACC and CUDA

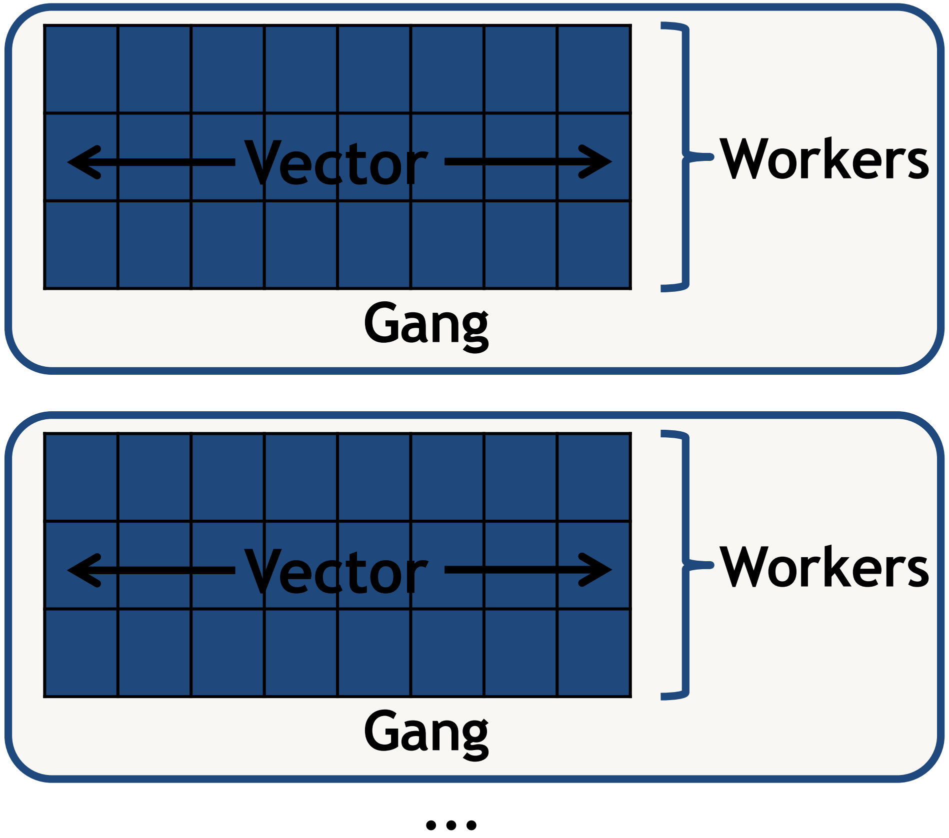
- Gangs are similar to CUDA blocks
  - Multiple gangs are executed independently from each other
  - No possibility to synchronize
  - Workers within a gang share local memory resources (e.g., shared caches) and can synchronize



Except for advanced cases, the organization in gangs, workers and vectors can be left completely to the compiler

Parallelism model – OpenACC and CUDA

- Different interpretations of workers and vectors on the various textbooks:
  - Each worker is a CUDA thread, executing same instructions on multiple data
  - Each worker is a warp and each vector element is a CUDA thread



Except for advanced cases, the organization in gangs, workers and vectors can be left completely to the compiler

OpenACC directives in C/C++

- In C/C++ OpenACC directives are specified as pragmas

```
#pragma acc <directive> <clauses>
```

The pragma is a standard method to provide information to the compiler

OpenACC pragma

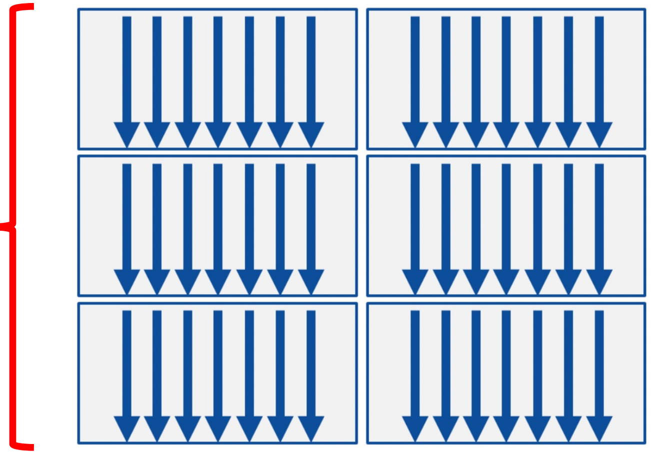
Specify commands to the compiler on how to parallelize

- Pragmas are referred to the subsequent region of code (loop or within brackets `{ }`)
- Pragmas are ignored by compilers not supporting them

`kernels` directive

- The `kernels` directive leaves the compiler to autonomously decide on how to parallelize the code

```
#pragma acc kernels
for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```



- The compiler analyzes loops to identify the parallelization scheme, to guarantee there is no loop dependency, …
- Organization in gangs is decided by the compiler

For the sake of simplicity details on workers and vectors are not reported

Compiling the code

- A specific compiler is required
  - NVIDIA one: https://developer.nvidia.com/openacc

- To compile:

```
nvc -fast -ta=tesla -Minfo=accel -o vectsum vectsum.c
```

Use all possible optimizations

Target NVIDIA device

Provide information on how code has been parallelized

- To target a multicore CPU: `-ta=multicore`

Compiling the code

- Compiler output on screen:

```
main:
    14, Loop is parallelizable
        Generating Tesla code
        14, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    14, Generating implicit copyout(C[:]) [if not already present]
        Generating implicit copyin(B[:],A[:]) [if not already present]
```

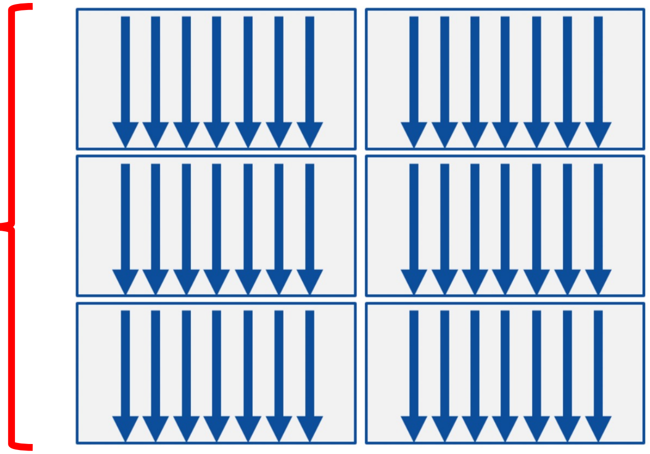
Code line numbers:

```
13  #pragma acc kernels
14  for(int i = 0; i < N; i++)
15    C[i] = A[i] + B[i];
```

`kernels` directive

- The annotated region of code may contain multiple nested loops

```
#pragma acc kernels
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    C[i*N+j] = A[i*N+ j] + B[i*N+ j];
```



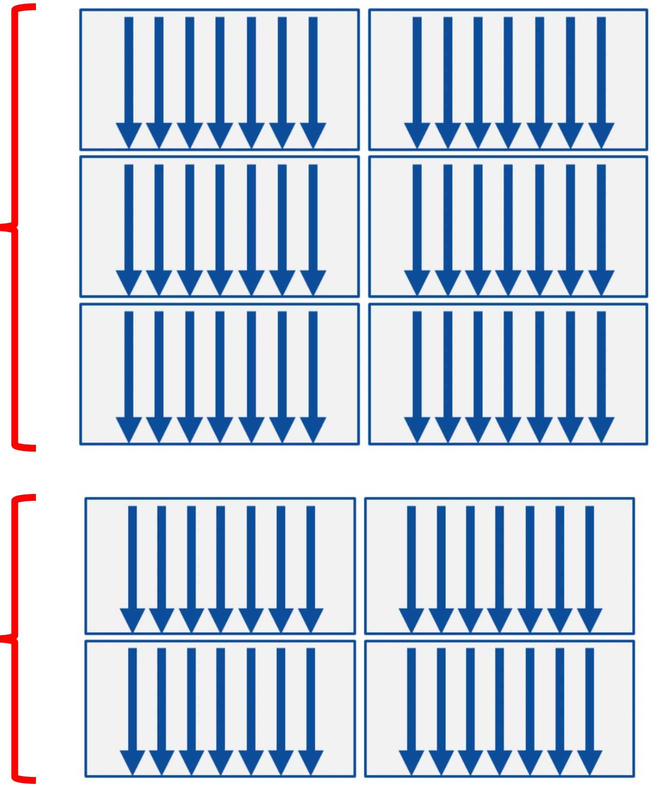
The compiler is able to analyze and manage also multiple nested loops

`kernels` directive

- The annotated region of code may contain multiple subsequent loops

```
#pragma acc kernels
{
  for(int i = 0; i < N; i++)
    C[i] = A[i] + B[i];


  for(int i = 0; i < M; i++)
    G[i] = D[i] + E[i];
}
```



Each loop within the same annotated region of code may be accelerated in different ways

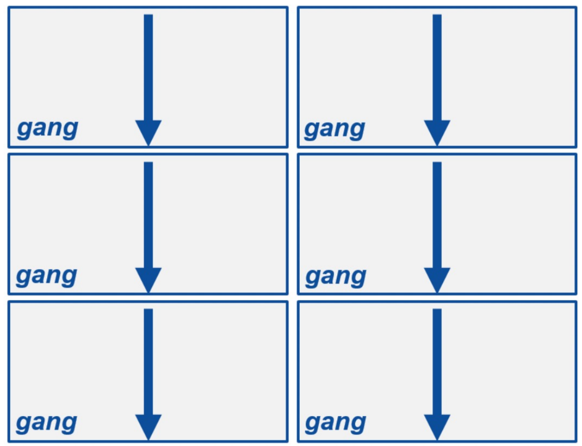
`parallel` directive

- The `parallel` directive marks a region of code for parallelization
- The `parallel` directive forces the compiler to follow programmer commands
  - **The programmer is responsible to correctly express parallelization**

- The basic behavior of the directive is to **redundantly parallelize** the code:

```
#pragma acc parallel
for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

- It creates 1 or more parallel gangs (each one with 1 worker and 1 vector element
- Each vector element executes redundantly the same code on the same data!



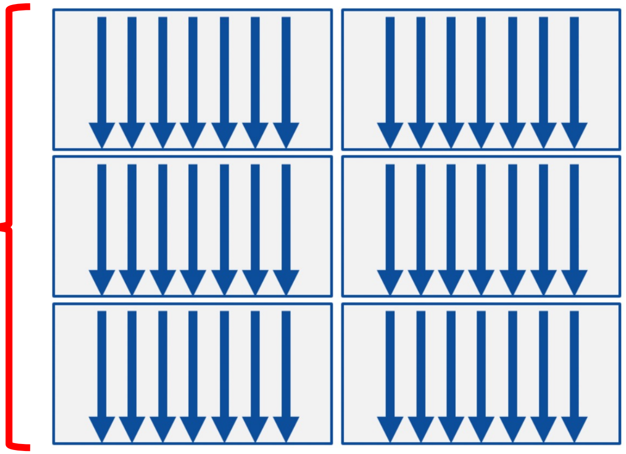
`parallel` directive

- The `parallel` directive is used together with the `loop` directive to parallelize loop execution

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

or

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
}
```



`parallel` directive

- Nested and sequential loops have to be explicitly managed

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
  #pragma acc loop
  for(int j = 0; j < N; j++)
    C[i*N+j] = A[i*N+ j]
                 + B[i*N+ j];
```

```
#pragma acc parallel
{
 #pragma acc loop
  for(int i = 0; i < N; i++)
    C[i] = A[i] + B[i];

  #pragma acc loop
  for(int i = 0; i < M; i++)
    G[i] = D[i] + E[i];
}
```

`parallel` directive is used only once at the beginning

Clauses

- Directives can be customized with several clauses:

```
int swap;
#pragma acc parallel loop private(swap)
for(int i = 0; i < N; i++){
  swap = A[i];
  A[i] = B[i];
  B[i] = swap;
}
```

- The `swap` variable has to be private per each single vector element to allow correct parallelization
- Multiple variables are listed separated by commas

```
#pragma acc parallel loop collapse(2)
for(int i = 0; i < N; i++)
  for(int j = 0; j < N; j++)
    C[i*N+j] = A[i*N+ j] + B[i*N+ j];
```

- A number of nested loops can be collapsed so that they are parallelized together

Clauses

- Directives can be customized with several clauses:

```
#pragma acc parallel loop seq
for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

- The loop is executed sequentially
- This clause may be useful in case of multiple nested loop to specify that some loop has not to be parallelized (coarsening strategy!)

```
int sum=0;
#pragma acc parallel loop reduction(+:sum)
for(int i = 0; i < N; i++)
  sum += A[i];
```

- Parallel reductions can be automatically performed
- Supported operators: +, *, max, min, &, |, &&, ||

Clauses

- Directives can be customized with several clauses:

```
#pragma acc parallel loop gang(16)
for(int i = 0; i < N; i++)
  #pragma acc loop worker
  for(int j = 0; j < N; j++)
    C[i*N+j] = A[i*N+ j] + B[i*N+ j];
```

- `gang`, `worker` and `vector` clauses may be used to specify how to parallelize loops
- Each clause optionally accepts the number of concurrent actors to be used
- `num_gangs(16)` is another clause that can be used to say how many gangs to use in a parallel section

Data movements between host and device

- In the case of discrete accelerator, the device memory is physically separate from the host one!
- The OpenACC compiler has to manage also data movements between host and device memory
  - It decides which data have to be moved based on variable reads and assignments
  - It may take conservative non-optimal decisions (too many data movements)

> - Same data cannot be accessed in a consistent way by the host and the device at the same time!
> - In the case of unified memory there is no guarantee that there is a single copy of the data shared between host and device

Data movements between host and device

- Compiler output on screen for the previous example:

```
main:
     14, Loop is parallelizable
         Generating Tesla code
         14, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     14, Generating implicit copyout(C[:]) [if not already present]
         Generating implicit copyin(B[:],A[:]) [if not already present]
```

Code line numbers:

```
13  #pragma directive kernels
14  for(int i = 0; i < N; i++)
15    C[i] = A[i] + B[i];
```

Data movements between host and device

- NVIDIA profiler can be used to analyze data movements to identify non-optimal behaviors
- If supported, CUDA managed memory may alleviate excessive memory transfers
  - Compile as follows:

```
nvc -fast -ta=tesla:managed -Minfo=accel vectsum.c
```

- Alternatively, data transfers can be managed by the programmed by means of the `data` directive

`data` directive

- The `data` directive specified data transfers in a region

```
#pragma acc data copyin(A[0:N], B[0:N]) copyout(C[0:N])
{
  #pragma acc kernels
  for(int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
}
```

- `copyin`: data are copied from the host to the device at the beginning of the parallel region and memory is freed at the end of the region
- `copyout`: device memory is allocated at the beginning of the parallel region and copied to the host memory at the end of the region
- `copy`: combines `copyin` and `copyout` behaviors
- `create`: allocates device memory
- Each array requires the index ranges to be specified

`data` directive

- The `data` directive can be combined with both the `kernels` and the `parallel` directives

```
#pragma acc kernels copyin(A[0:N], B[0:N]) copyout(C[0:N])
for(int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

Further features

- Further directives and clauses. E.g.:
  - `#pragma acc kernels async(3)`
    - Asynchronously runs the subsequent parallel region assigning an id equal to 3
  - `#pragma acc wait(3)`
    - Stops the host waiting for the end of parallel region with id equal to 3

- Functions (including `openacc.h` header). E.g.:
  - `acc_async_wait(3);`
    - Is the same as above
  - `acc_set_device_type(acc_device_nvidia);`
    - Sets at runtime the usage of an NVIDIA GPU

A complete example

- Let’s consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

Example taken from: D. B. Kirk, W.-m. W. Hwu, **Programming Massively Parallel Processors: A Hands-on Approach, Chapter 19, 3rd edition**

A complete example

- Let’s consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

`A` is matrix representing the current temperature of the single (discretized) positions in the 2D surface

A complete example

- Let's consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

The temperature of the surface is computed iteratively for `iter_max` iterations

A complete example

- Let’s consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
```

A

1.0	1.0	1.0	1.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Anew

1.0	1.0	1.0	1.0
0.0	0.25	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

- The temperature of each single point is computed as the average of the temperature of the 4 neighbor points during previous iteration
- All points of `A` are iterated

A complete example

- Let’s consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

- `A_new` is used to store new temperatures
- `A_new` is copied back to `A`

A complete example

- Let's consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

- The maximum difference between old and new temperature of a single point is computed
- If lower than a given constant computation is interrupted

A complete example

- Let’s consider the Jacopi iterative method that solves the Laplace equation to compute the heating transfer on a 2D surface

```
while ( err > tol && iter < iter_max ) {
  err=0.0;
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

- `while` loop has data dependencies among iterations -> it cannot be parallelized
- Iterations in each of the two nested `for` loops are independent -> they can be parallelized
- `err` is computed by means of a reduction
- `A` has to be transferred to the device at the beginning of the main and back to the host at the end
- `Anew` is used only in the device

A complete example

- Solution with OpenACC directives:

```
#pragma acc data create(Anew[0:n][0:m]) copy(A[0:n][0:m])
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc parallel loop reduction(max:err) collapse(2)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j -1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
#pragma acc parallel loop collapse(2)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

References

- Slides mainly based on:
  - D. B. Kirk, W.-m. W. Hwu, **Programming Massively Parallel Processors: A Hands-on Approach**, **Chapter 19** **3rd edition**
  - J. Chen, M. Grossman, T. McKercher, **Professional Cuda C Programming**, **Chapter 8**
  - NVIDIA Deep Learning Institute, **Fundamentals of Accelerated Computing with OpenACC** https://courses.nvidia.com/courses/course-v1:DLI+C-AC-03+V1/

The NVIDIA voucher can be used also to attend this course. Highly suggested!