Directive Based Parallel programming on GPU

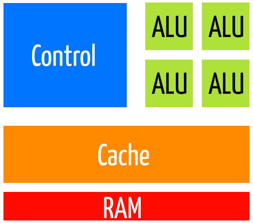
Orian Louant
`orian.louant@uliege.be`

November 10, 2020

CPU vs GPU

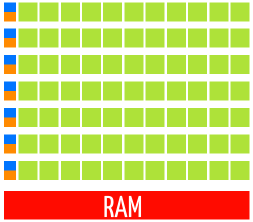
CPU

- Optimized for low-latency access to cached data
- Control logic for out-of-order and speculative execution



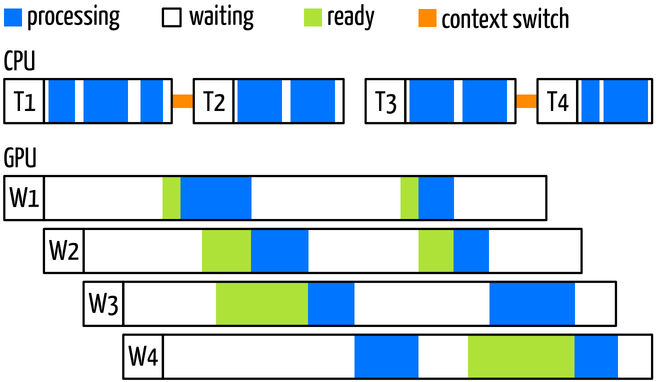
GPU

- Optimized for data-parallel, high-throughput calculation
- Tolerance for memory latency
- Large part of the die dedicated to the computation



CPU vs GPU

- CPU cores are optimized to minimize latency between operations
- GPU minimize latency between operations by scheduling multiple threads bundle: Wraps (NVIDIA) or Wavefronts (AMD)



Programming Models for GPU

While CPU requires coarse-grained parallelism, GPU requires fine grained one. You have to divide your calculation in small computational kernels working on small pieces of data. To expose this fine-grained parallelism, new programming models have been created:

- **Vendor specific:** NVIDIA CUDA and AMD HIP
- **Cross-platform:** OpenCL, SYCL
- **High-level frameworks:** Thrust, Kokkos, ...

With these models, existing codes need to be rewritten or refactored.

Directive Based Models for GPU

Instead of writing new code, why not use the existing code and add compiler directives to offload execution to the GPU.

- **OpenACC:** specifically designed to target accelerator device
- **OpenMP:** designed for multicore CPUs but since version 4.0, can also target accelerator device

In these models, most of the work to port the code to the GPU is done by the compiler. The developer only manage the high-level representation.

Directive Based Models for GPU

Directives are special compiler instruction which constitutes hints to the compiler on how to process the source. In our case, how to transform the code so that it can run on a GPU.

OpenACC

```
#pragma acc directive-name [clause-list]
  structured-block
```

```
!$acc directive-name
    [clause-list]
  block-of-code
!$acc end directive-name
```

OpenMP

```
#pragma omp directive-name [clause-list]
  structured-block
```

```
!$omp directive-name
    [clause-list]
  block-of-code
!$omp end directive-name
```

Directive Based Models for GPU

Programming GPU using directives will not guarantee you the same performance as using the native solution of the hardware vendor. However, you might have good reason to use the directive model.

- You have a large code base and translating the entire code to CUDA/HIP will take too much time
- You want a quick away use GPU with part of your code that is not yet translated to CUDA/HIP
- Your plan is to use LUMI and your code is in Fortran

OpenMP Offload Execution Model

SAXPY with OpenMP Directives

As an illustration, we will use the saxpy example: $y_i = \alpha x_i + y_i$

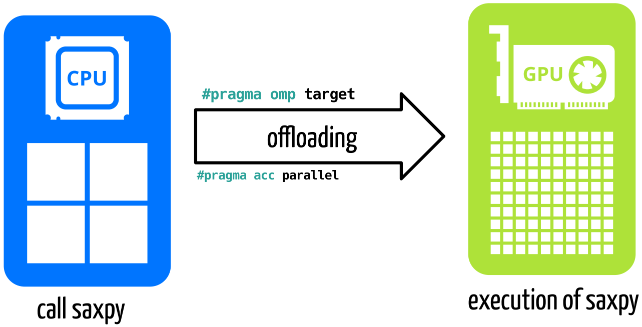
```c
void saxpy(int n,
           float a,
           float* x,
           float* restrict y) {

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

// [...]

saxpy(1<<20, 2.0, x, y);
```

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

  !$omp  target teams distribute &
  !$omp& parallel do
    do i = 1,n
      y(i) = a*x(i) + y(i)
    end do
  !$omp  end target teams distribute &
  !$omp& parallel do
end subroutine

! [...]

call saxpy(2**20, 2.0, x, y)
```

Offloading with Directives

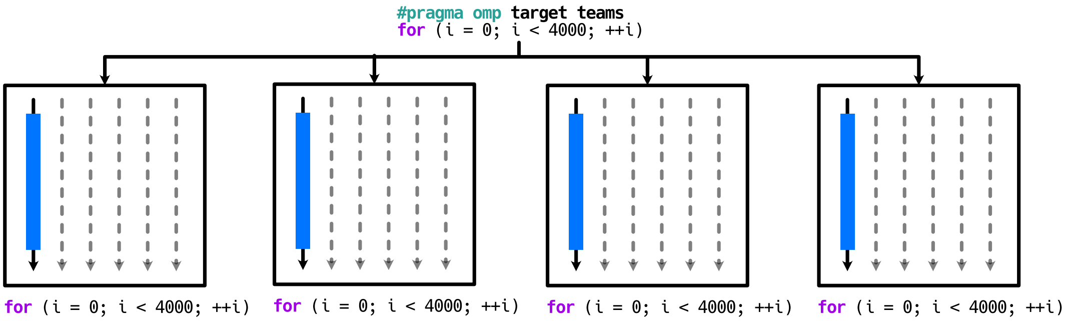
The first part of the directives used in the saxpy example instruct the compiler that the next block of code should be offloaded to the GPU.

```
#pragma omp target
```



Execution Model: Teaming

When using the `teams` directive, a league of teams is created. The master threads of each team execute redundantly all the iterations of the loop.

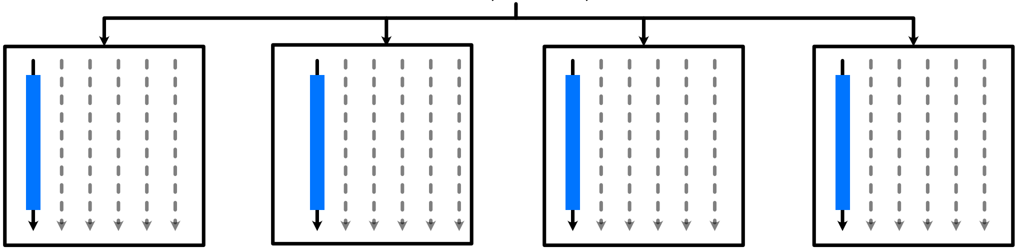


#pragma omp target teams
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)

Execution Model: Distributing to the Teams

When using the `teams distribute` directive, the iterations of the loop are distributed to the master threads of the teams.

```
#pragma omp target teams distribute
for (i = 0; i < 4000; ++i)
```



```
for (i = 0; i < 1000; ++i)    for (i = 1000; i < 2000; ++i)    for (i = 2000; i < 3000; ++i)    for (i = 3000; i < 4000; ++i)
```

Teams and Distribute Constructs

The **teams** construct creates a league of thread teams where the master thread of each team executes the block of code that follows the directive.

```
#pragma omp teams
  structured-block
```

```
!$omp teams
  block
!$omp end teams
```

The **distribute** construct specifies that the iterations of one loop should be distributed across the master threads of all teams.

```
#pragma omp distribute
  for-loop
```

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

Teams and Distribute Constructs

The number of teams created is implementation dependant but, you can specify it using the `num_teams` clause.

```
#pragma omp teams num_teams(int-expr)
  structured-block
```

```
!$omp teams num_teams(int-expr)
  block
!$omp end teams
```

The number of teams created will then be less than or equal to the `int-expression` given as argument of the clause. Another option is to specify the maximun number of threads participating in the teams with the `thread_limit` clause.

```
#pragma omp teams
    thread_limit(int-expr)
  structured-block
```

```
!$omp teams thread_limit(int-expr)
  block
!$omp end teams
```

Teams and Distribute Constructs

In order to retrieve the number of teams, you can use the **omp_get_num_teams** function that returns the number of teams in the current teams region, or 1 if called from outside of a teams region.

```c
int omp_get_num_teams();
```

```fortran
integer function omp_get_num_teams()
```

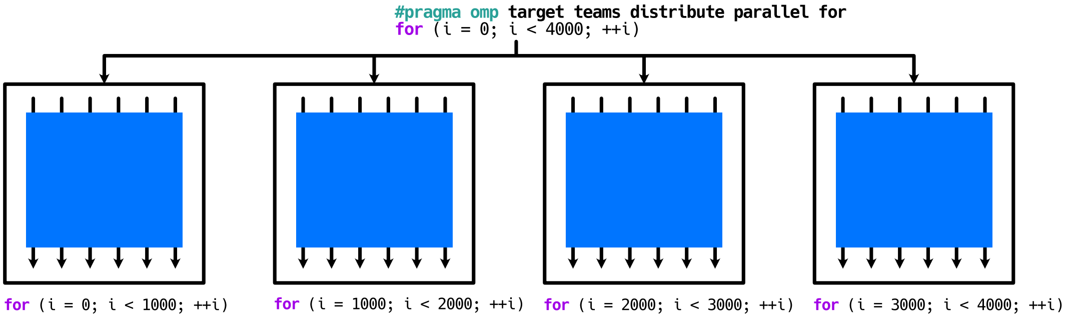
To get the team number of a thread, use the **omp_get_team_num** function. The team number is an integer between 0 and the value returned by **omp_get_num_teams** - 1.

```c
int omp_get_team_num();
```

```fortran
integer function omp_get_team_num()
```

Execution Model: Distributing to the Threads

When adding the `parallel for` directive, the iterations of the loop assigned to the teams are then distributed to the threads in these teams.



#pragma omp target teams distribute parallel for
for (i = 0; i < 4000; ++i)
for (i = 0; i < 1000; ++i)
for (i = 1000; i < 2000; ++i)
for (i = 2000; i < 3000; ++i)
for (i = 3000; i < 4000; ++i)

Distribute Parallel Loop Constructs

The **`distribute parallel for`** construct specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams.

```
#pragma omp distribute parallel for
  for-loop
```

```
!$omp distribute parallel do
  do-loop
!$omp end distribute parallel do
```

As we have seen in the saxpy example, this construct can be used as a combined construct with **`teams`** to create the teams of threads and the share the work among the teams and as well as the threads in the teams.

```
#pragma omp teams distribute parallel for
  for-loop
```

```
!$omp teams distribute parallel do
  do-loop
!$omp end teams distribute parallel do
```

Distribute Parallel Loop Constructs

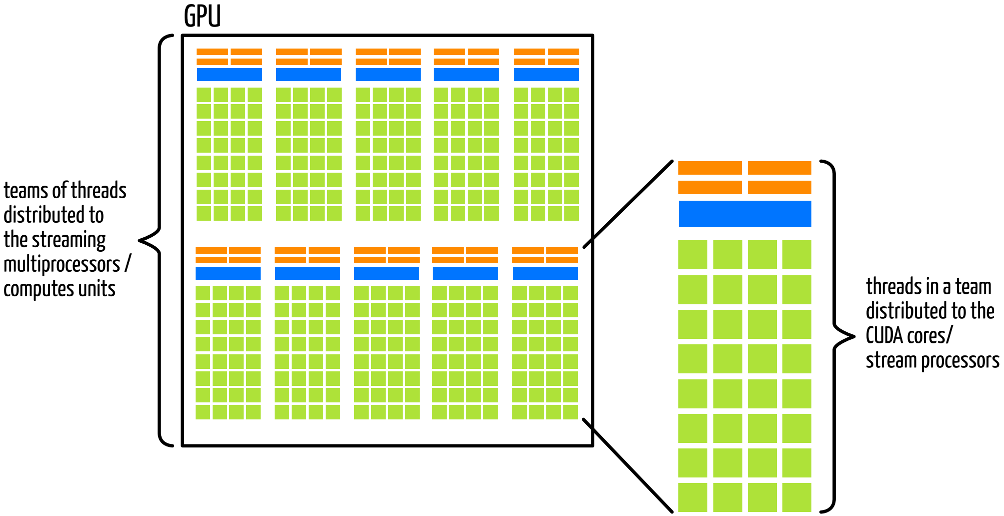
Distribution of the loop iterations using the **teams distribute** and **parallel for** can used to parallelize two levels of a loop nest.

```c
#pragma omp teams distribute
for (int i = 0; i < M; ++i) {
  #pragma omp parallel for
  for (int j = 0; j < N; ++j)
    // do something
}
```

```fortran
!$omp teams distribute
  do i = 1,M
    !$omp parallel for
      do j = 1,N
      ! do something
      end do
    !$omp end parallel for
  end do
!$omp end teams distribute
```

Here, the iterations of the i-loop are distributed to the gangs. Then, the iterations of the j-loop are distributed to the threads in the gangs.

Execution Model: Hardware


GPU
teams of threads distributed to the streaming multiprocessors / computes units
threads in a team distributed to the CUDA cores/ stream processors

OpenACC Execution Model

SAXPY with OpenACC Directive

If we go back to our saxpy example, using OpenACC, the code will become:

```c
void saxpy(int n,
           float a,
           float* x,
           float* restrict y) {

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

// [...]

saxpy(1<<20, 2.0, x, y);
```

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

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

! [...]

call saxpy(2**20, 2.0, x, y)
```

Offloading with OpenACC Directives

In addition to the `parallel` construct, OpenACC also include the `kernels` contruct.

```
#pragma acc kernels
```

- With the `kernels` construct, the programmer identifies a region of code that may contain parallelism. It gives the compiler more freedom to find and map parallelism according to the requirements of the target accelerator. Similar to an automatic parallelization.
- The `parallel` construct is more explicit. It identifies a region of code that will be parallelized across gangs. Needs to be paired with a `loop` directive for work-sharing.

SAXPY with OpenACC Directives

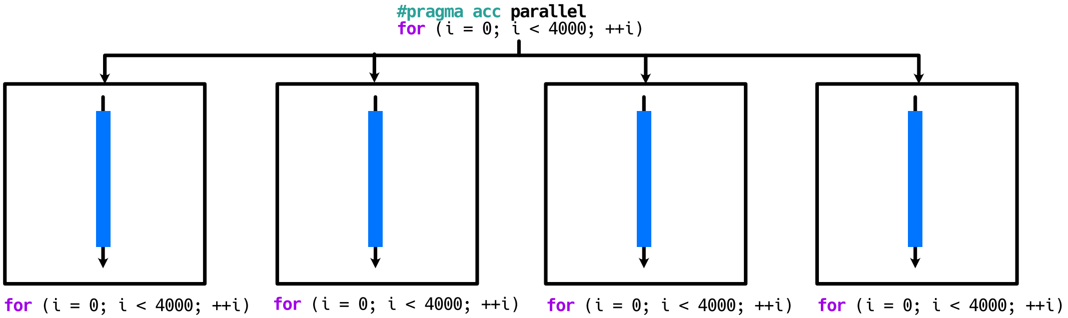
So we have the choice to use either the `parallel` construct and specify by ourself the parallelism, or let the compiler do the work with the `kernels` construct.

```c
void saxpy(int n,
           float a,
           float* x,
           float* restrict y) {

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

// [...]

saxpy(1<<20, 2.0, x, y);
```

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

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

! [...]

call saxpy(2**20, 2.0, x, y)
```

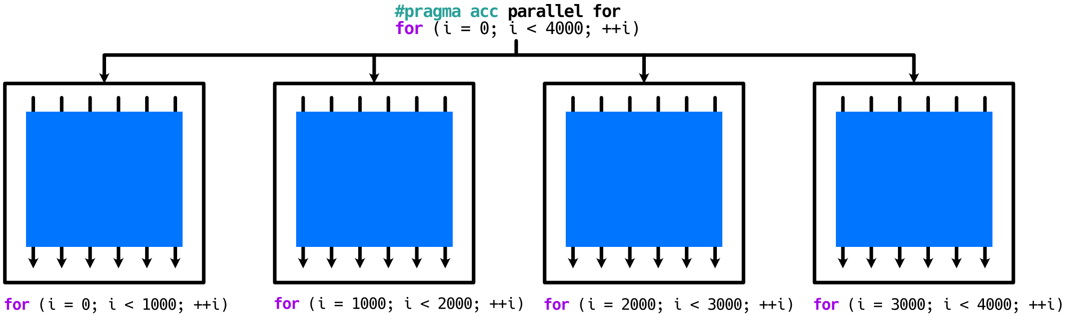
Execution Model: Gang Redundant Mode

When using the `parallel` directive, one or more gangs of workers are created to execute the region on the GPU. Each gang begins executing the code in the structured block in gang-redundant mode. This means that code within the parallel region will be executed redundantly by all gangs.


#pragma acc parallel
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)
for (i = 0; i < 4000; ++i)

Execution Model: Loop Construct

When using the `loop` directive, we instruct the compiler that the iterations of the loop that follows the directive should be distributed among the gangs and the workers of the gangs.



#pragma acc parallel for
for (i = 0; i < 4000; ++i)
for (i = 0; i < 1000; ++i)
for (i = 1000; i < 2000; ++i)
for (i = 2000; i < 3000; ++i)
for (i = 3000; i < 4000; ++i)

Execution Model: The Parallel and Loop Constructs

In summary, to create a kernel and the gangs we use the `parallel` construct.

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

```
!$acc parallel
  block
!$acc end parallel
```

Then to share the iterations of a loop between the gangs and the workers in the gangs, we use the `loop` construct

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

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

This can be done in one step using the combined `parallel loop` construct

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

```
!$acc parallel loop
  do-loop
!$acc end loop
```

Execution Model: The Three Levels of Parallelism

OpenACC exposes three levels of parallelism gang, worker and vector. A gang is a group of workers which execute vector operations of a certain length.

	NVIDIA	AMD
`gang`	Thread Block	Workgroup
`worker`	Warp	Wavefront
`vector`	Thread	Work item/Thread

So that, the number of threads in a gang can be computed as

$$N_{\text{threads}} = L_{\text{vector}} \cdot N_{\text{workers}}$$

Execution Model: The Three Levels of Parallelism

The number of gangs, worker and vector length can be specified using optional clauses to the `parallel` construct.

```
#pragma acc parallel num_gangs(int-expr)   \
                     num_workers(int-expr) \
                     vector_length(int-expr)
```

```
!$acc  parallel num_gangs(int-expr)     &
!$acc&          num_workers(int-expr)   &
!$acc&          vector_length(int-expr)
```

- `num_gangs` defines the number of parallel gangs that will execute the parallel region.
- `num_workers` fefines the number of workers within each gang that will be active after a gang transitions from worker-single mode to worker-partitioned mode
- `vector_length` defines the number of vector lanes that will be active after a worker transitions from vector-single mode to vector-partitioned mode.

Execution Model: The Three Levels of Parallelism

In our saxpy example, we let the compiler choose the level of parallelism automatically, but the programmer can choose to specify the level of parallelism using optional clauses to the `loop` construct.

```c
#pragma acc loop gang
for (int i = 0; i < M; ++i) {
  #pragma acc loop vector
  for (int j = 0; j < N; ++j)
  // do something
}
```

```fortran
!$acc loop gang
  do j = 1,N
    !$acc loop vector
      do i = 1,M
      ! do something
      end do
    !$acc end loop
  end do
!$acc end loop
```

A `gang` loop may not appear inside `worker` loop, which may not appear within a `vector` loop

Compilers Support

Compilers Support

We are still at an early stage of the development of directive based GPU programming. Compiler support and quality of the implementation for OpenACC and OpenMP offloading may vary.

OpenACC

- **GCC:** NVIDIA and AMD hardware. Poor performance in some cases, but improving.
- **NVC/PGI:** Only for NVIDIA hardware. Best implementation.

OpenMP

- **Clang:** Support for NVIDIA and AMD hardware. Best implementation.
- **GCC:** Support for NVIDIA and AMD hardware. Same comments that for OpenACC

Compilers Support

- **NVIDIA:** NVC/PGI is your best option for OpenACC. You can install this compiler throught the NVIDIA HPC SDK. Also available on dragon2 (module pgi/19.10). Use Clang for OpenMP.
- **AMD:** You are targeting LUMI? Then go for OpenMP.

On LUMI, you will have access the AMD version of the Clang and Flang compilers to target GPU with OpenMP. In addition, the Cray Compiling Environment (CCE) which also support offloading to the GPU through OpenMP use will be available.

Compiling an OpenMP Application

An OpenMP application with offloading targeting GPU can be compiled using GCC or Clang. These two compilers have support for AMD and NVIDIA hardware.

Compiling with Clang/Flang

```
clang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -o <exe_out> <source.c>
clang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -o <exe_out> <source.c>

flang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -o <exe_out> <source.f90>
flang -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -o <exe_out> <source.f90>
```

Compiling with GCC/GFortran

```
gcc -fopenmp -foffload=nvptx-none -o <exe_out> <source.c>
gcc -fopenmp -foffload=amdgcn-amdhsa -o <exe_out> <source.c>

gfortran -fopenmp -foffload=nvptx-none -o <exe_out> <source.f90>
gfortran -fopenmp -foffload=amdgcn-amdhsa -o <exe_out> <source.f90>
```

Compiling an OpenACC Application

An OpenACC application targeting GPU can be compiled using GCC or the NVIDIA compiler. GCC has support for AMD and NVIDIA hardware while NVC target only NVIDIA hardware.

Compiling with GCC/GFortran

```
gcc -fopenacc -foffload=nvptx-none -o <exe_out> <source.c>
gcc -fopenacc -foffload=amdgcn-amdhsa -o <exe_out> <source.c>

gfortran -fopenacc -foffload=nvptx-none -o <exe_out> <source.f90>
gfortran -fopenacc -foffload=amdgcn-amdhsa -o <exe_out> <source.f90>
```

Compiling with NVC/NVFortran

```
nvc -acc -Minfo=accel -o <exe_out> <source.c>
pgcc -acc -Minfo=accel -o <exe_out> <source.c>

nvfortran -acc -Minfo=accel -o <exe_out> <source.f90>
pgf90 -acc -Minfo=accel -o <exe_out> <source.f90>
```

Data Management

Data Management

In the saxpy example, we have left aside the very important topic of data management. This management of the data between the host and the device is the programmer's responsibility.

- You must make sure that all the necessary data for a computation is available on the GPU before entering the compute region
- You must make sure to transfer the processed data back to the CPU memory if needed

Data Management

The following code compiled with clang will crash because the **x** and **y** arrays are not present in the GPU memory

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

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

The OpenACC version may be fine with the NVIDIA compiler as it may do automatic data management but it will crash with GCC.

Data Management

In order for the code to run on the GPU, we need to copy the `x` and `y` arrays in the GPU memory. With OpenMP this is done using the `target data` construct and the `map` clause.

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

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

Similarly, OpenACC offers the `data` construct to copy data to and from the GPU memory.

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

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

Data Management

OpenMP copy of data in and from the GPU memory this is done using the `target data` construct and the `map` clause.

```
#pragma omp target data [map-clauses]
  structured-block
```

```
!$omp target data [map-clauses]
  block
!$omp end target data
```

Similarly, OpenACC offers the `data` construct to copy data to and from the GPU memory.

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

```
!$acc data [clauses]
  block
!$acc end data
```

Data Management

We can describe the data management using 4 categories:

- used on the GPU but not modified
- modified on the GPU
- used and modified on the GPU
- created and only used on GPU

In addition, we can also consider the lifetime of the data:

- span only scope
- span multiple scopes or source files

Data Management

In OpenACC data operation are described by clauses while in OpenMP this is described by a type to the `map` clauses.

OpenACC	OpenMP	Description
`copyin`	**`to:`**	copy data to the GPU
`copyout`	**`from:`**	copy data from the GPU
`copy`	**`tofrom:`**	copy datato and from the GPU
`create`	**`alloc:`**	allocate data on the GPU

Data Management

The `map` clause can also be used with the `target` construct. This is useful when the data lifetime match the compute region.

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

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

Similarly, with OpenACC, you can use the data clauses with the `parallel` construct.

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

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

Data Management

In C/C++, for dynamically allocated arrays, you have to specified the lower bound and number of elements of the array.

```
#pragma omp target data map(tofrom: x[0:n])
```

```
#pragma acc data copy(x[0:n])
```

In Fortran, array shape information is already embedded in the data type. But indexing may be required in order to map an array subrange. In C/C++ subranges are defined as **`start_idx:len`** while in Fotran it's defined as **`start_idx:end_idx`**.

```
#pragma omp target data map(tofrom: x[2:n-2])
#pragma acc data copy(x[2:n-2])
```

```
!$omp target data map(tofrom: x[3:n])
!$acc data copy(x[3:n])
```

Data Management

So far we have discussed what is called **structured data region**, this means that the data will be:

- copied in (or created) at entry of the region
- copied out (or deallocated) at the exit.

Another option is to use **unstructured data region** where data might span multiple scopes (including multiple source files). This type data region is created with:

```
#pragma omp target enter data [map-clauses]

#pragma acc enter data [clauses]
```

```
!$omp target enter data [map-clauses]

!$acc enter data [clauses]
```

and the end of the region is defined as follows

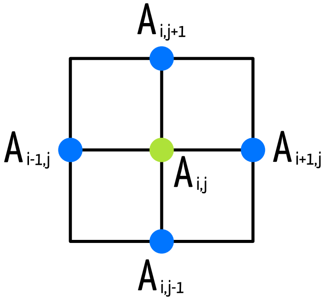
```
#pragma omp target exit data [map-clauses]

#pragma acc exit data [clauses]
```

```
!$omp target exit data [map-clauses]

!$acc exit data [clauses]
```

Case Study: Laplace Heat Equation

Jacobi Iteration

- Iteratively converge to the solution by computing new values at each point from the average of the neighbouring points.
- In our example we will solve the Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A^{k+1}_{i,j} = \frac{A^k_{i-1,j} + A^k_{i+1,j} + A^k_{i,j-1} + A^k_{i,j+1}}{4}$$

Jacobi Iterations: Serial Code

We start from the serial code for the Laplace steady state heat equation, and we will focus our attention on the main computational part:

```
while (error > tol && niter < niter_max) {
  error = 0.0;

  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i+j*m] = Anew[i+j*m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);

  niter++;
}
```

Jacobi Iteration

We will consider a 2D grid of size 4096x4096. The execution times have been measured on a GPU node of the Dragon2 CÉCI cluster: Xeon Gold 6126 12-core CPU and NVIDIA Tesla V100 GPU.

- **OpenMP:** Clang version 10 compiler
- **OpenACC:** PGI version 19.10 compiler

Jacobi Iterations: CPU Threaded Version

The first step is to consider parallelization of the CPU using OpenMP in order to compare the CPU parallel version with the GPU version.

```
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma omp parallel for reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma omp parallel for
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i+j*m] = Anew[i+j*m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);

  niter++;
}
```

The Reduction Clause

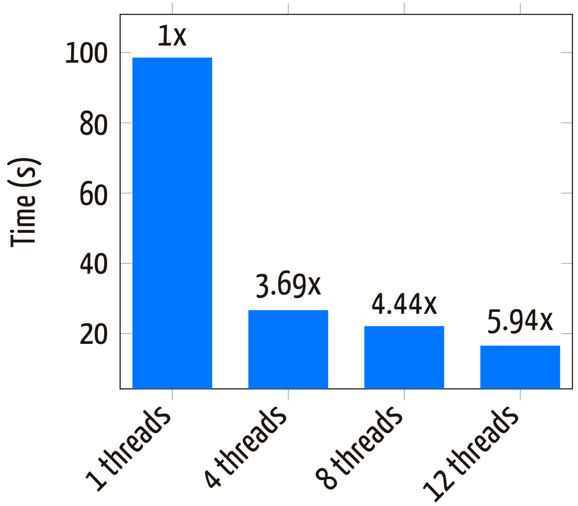
Data reduction is reducing a set of numbers into a smaller set of numbers. For example, summing elements of an array or find the min/max value in an array. The `reduction` clause avoid data races when summing or combining values. It can be used with the `parallel`, `teams` and work-sharing contructs.

```
reduction(op:list)
```

`op` is an operator:

- Arithmetic reductions: `+ * - max min`
- Logical operator reductions: `& && | ||`

Laplace Heat: CPU Using Threads



- We get a reasonable speedup using 4 threads
- For more that 4 threads, we get a speedup but, not as good as previously
- We are limited by memory bandwidth

Laplace Heat: Outer Loop on GPU

For the GPU version, we will explore two solutions: the first is to distribute the iterations of the outer loop and let the inner loop untouched.

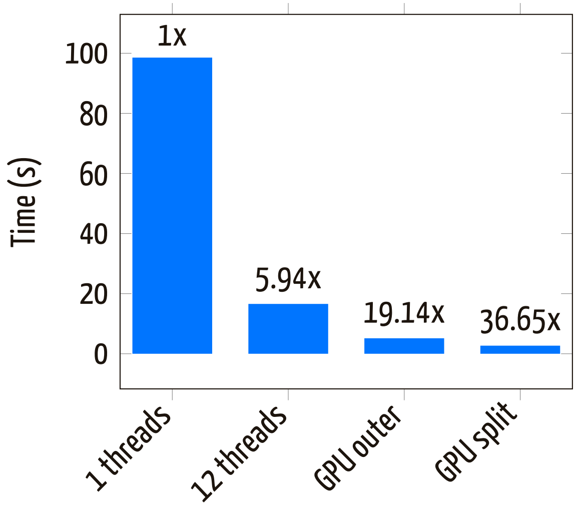
```c
#pragma omp target data map(to: Anew[0:n*m]) map(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma omp target teams distribute parallel for reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma omp target teams distribute parallel for
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: Split the Loops on GPU

For the GPU version, we will explore two solutions: the second is to distribute the iterations of the outer to the teams and inner loop to the threads in the teams.

```
#pragma omp target data map(to: Anew[0:n*m]) map(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma omp target teams distribute reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    #pragma omp parallel for reduction(max:error)
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma omp target teams distribute
  for (int j = 1; j < n-1; ++j) {
    #pragma omp parallel for
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: CPU vs GPU (OpenMP)



- Parallelizing the outer loop leads to a 3.2x speedup with respect to the CPU with 12 threads
- Worksharing the two loops leads to a 6.1x speedup with respect to the CPU with 12 threads

Laplace Heat: OpenACC

Using OpenACC, we will test the two solutions previously tested with OpenMP: the first is to distribute the iterations of the outer loop.

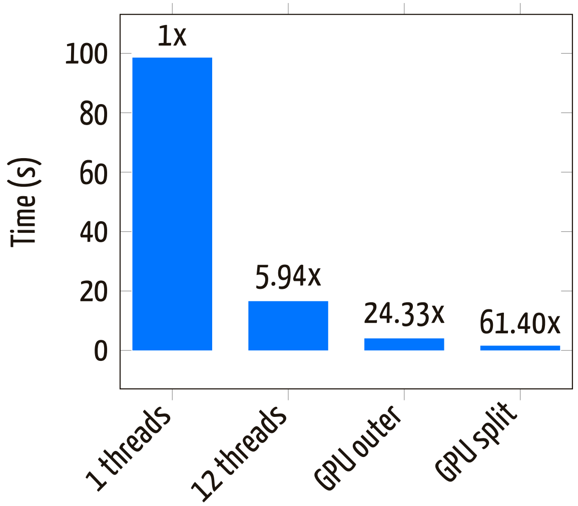
```
#pragma acc data copyin(Anew[0:n*m]) copy(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma acc parallel loop reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma acc parallel loop
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: OpenACC

Using OpenACC, we will test the two solutions previously tested with OpenMP: the second is to distribute the iterations of the two loops.

```
#pragma acc data copyin(Anew[0:n*m]) copy(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma acc parallel loop reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    #pragma acc loop reduction(max:error)
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma acc parallel loop
  for (int j = 1; j < n-1; ++j) {
    #pragma acc loop
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: CPU vs GPU (OpenACC)



- Parallelizing the outer loop leads to a 4.1x speedup with respect to the CPU with 12 threads
- Worksharing the two loops leads to a 10.3x speedup with respect to the CPU with 12 threads

Loop collapsing

Another idea we can explore is to collapse the loop nest into one big loop in order to increase parallelism. This can be done using the `collapse` clause in both OpenACC and OpenMP.

The `collapse` clause, collapse the iterations of the n-associated loops to which the clause applies into one larger iteration space. This clause can only apply on tightly nested loops, meaning that there is no code between the loops.

```
#pragma omp for collapse(n)
  nested-for-loops

#pragma omp loop collapse(n)
  nested-for-loops
```

```
!$omp do collapse(n)
  nested-do-loops

!$acc loop collapse(n)
  nested-do-loops
```

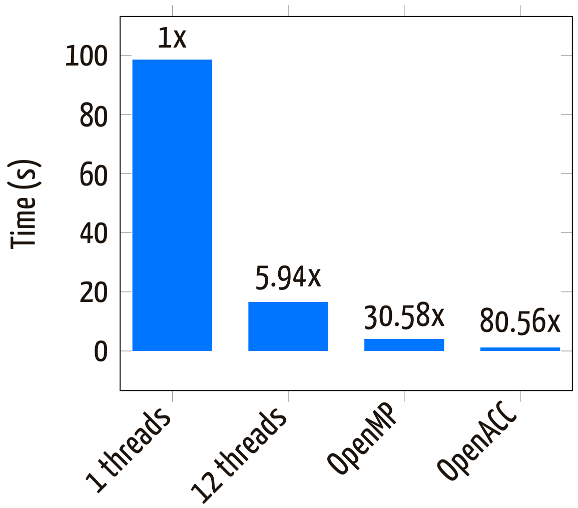
Laplace Heat: Loop collapse

```c
#pragma acc data copyin(Anew[0:n*m]) copy(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma acc parallel loop collapse(2) reduction(max:error)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma acc parallel loop collapse(2)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: Loop collapse

```
#pragma omp target data map(to: Anew[0:n*m]) map(A[0:n*m])
while (error > tol && niter < niter_max) {
  error = 0.0;

  #pragma omp target teams distribute parallel for reduction(max:error) collapse(2)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; ++i) {
      unsigned int idx = i + j * m;
      Anew[idx] = 0.25 * ( A[idx+1] + A[idx-1] + A[idx-m] + A[idx+m]);
      error = fmax(error, fabs(Anew[idx] - A[idx]));
    }
  }

  #pragma omp target teams distribute parallel for collapse(2)
  for (int j = 1; j < n-1; ++j) {
    for (int i = 1; i < m-1; i++ ) {
      A[i + j * m] = Anew[i + j * m];
    }
  }

  if(niter % 100 == 0) printf("%5d, %0.6f\n", niter, error);
  niter++;
}
```

Laplace Heat: CPU vs GPU (Loop Collapse)



- Loop collapsing with OpenMP improves the performance compared to parallelization of the outer loop but is slower than distributing the loops between teams and threads.
- Loop collapsing with OpenACC improves the performance with a 13.6x speedup with respect to the CPU with 12 threads

Laplace Heat: The Final Word

The result for the Laplace heat code can be summarized as follows:

	OpenACC	OpenMP
Speedup vs. serial	80.7x	36.7x
Speedup Socket (CPU) to socket (GPU)	13.6x	6.1x

- OpenACC with the PGI compiler display the best performance
- Targeting GPU with OpenMP is still only at the beginning, implementations may lack optimization
- After some more investigation, it turns out that it is the reduction that is the main bottleneck of the OpenMP code

Asynchronous Execution

Asynchronous vs. Synchronous Execution

- Without additional clauses, OpenMP and OpenACC directives are blocking: the CPU triggers the kernel launch or data transfer and wait until completion.
- Operations are inserted into the default execution queue. Items in this queue are executed in the order in which they have been inserted.

However, some algorithms have independent pieces of work which can be executed in any order and/or simultaneously.

- Thinks does not always require to execute in the order they appear in the source code.
- Data movement and computation can be overlapped

Mandelbrot Set

As an illustration, let's consider the Mandelbrot set. As we will call the **`mandelbrot`** function from the GPU, we need to declare it to be offloaded so that the compiler generates code that can be run on the GPU.

```
#pragma omp declare target
unsigned char mandelbrot(int px, int py) {
  const double x0 = XMIN + px * DX;
  const double y0 = YMIN + py * DY;

  double x = 0.0;
  double y = 0.0;

  int i;
  for(i = 0; (x * x + y * y) < 4.0
    && i < MAX_ITERS; i++) {
    double xtemp = x * x - y * y + x0;

    y = 2 * x * y + y0;
    x = xtemp;
  }

  return (double)MAX_COLOR*i/MAX_ITERS;
}
#pragma omp end declare target
```

```
#pragma acc routine seq
unsigned char mandelbrot(int px, int py) {
  const double x0 = XMIN + px * DX;
  const double y0 = YMIN + py * DY;

  double x = 0.0;
  double y = 0.0;

  int i;
  for(i = 0; (x * x + y * y) < 4.0
    && i < MAX_ITERS; i++) {
    double xtemp = x * x - y * y + x0;

    y = 2 * x * y + y0;
    x = xtemp;
  }

  return (double)MAX_COLOR*i/MAX_ITERS;
}
```

The Declare and Routine Directives

The OpenMP `declare target` directive specifies that variables and functions are mapped to a device so that these variables and functions can be accessed or executed on the device.

```
#pragma omp declare target
  var-or-function-declaration
#pragma omp end declare target
```

```
subroutine foo()
  !$omp declare target
end subroutine

!$omp declare target (list)
```

Concerning OpenACC, mapping a function to a device is done using the `routine` directive.

```
#pragma acc routine
  function-declaration
```

```
subroutine foo()
  !$acc routine
end subroutine
```

An optional clause that specifies that the function may contain a certain level (`gang`, `worker`, `vector` or `seq`) of parallelism may be added to this directive.

Mandelbrot Set

We can divide the computation in blocks along the y-axis and launched it as **`num_blocks`** kernels. The copy of the result from the GPU to the CPU memory is also done by blocks using the **`update`** (OpenACC) construct and **`target update`** (OpenMP).

```
#pragma omp target data \
map(alloc: image[0:WIDTH*HEIGHT])}
for(int block = 0; block < num_blocks; ++block) {
  int start = block * (HEIGHT/num_blocks);
  int end   = start + (HEIGHT/num_blocks);

  #pragma omp target teams distribute parallel for \
  collapse(2)
  for (int y = start; y < end; y++) {
    for (int x = 0; x < WIDTH; x++) {
      image[x + y * WIDTH] = mandelbrot(x, y);
    }
  }

  #pragma omp target update \
  from(image[block*block\_size:block\_size])
}
```

```
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; ++block) {
  int start = block * (HEIGHT/num_blocks);
  int end   = start + (HEIGHT/num_blocks);

  #pragma acc parallel loop collapse(2)
  for (int y = start; y < end; y++) {
    for (int x = 0; x < WIDTH; x++) {
      image[x + y * WIDTH] = mandelbrot(x, y);
    }
  }

  #pragma acc update \
  self(image[block*block\_size:block\_size])
}
```

The Update Directive

The OpenMP `target update` directive makes the list items in the device data environment consistent with the original list items by copying data between the CPU and the GPU. The `from` clause allows copy from the GPU to the CPU and the `to` clause copy from the CPU to the GPU.

```
#pragma omp target update from|to(list)
```

```
!$omp target update from|to(list)
```

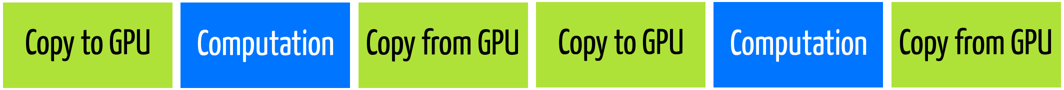
The same operation can be performed with OpenACC using the `update` directive. The `host` clause allows copy from the GPU to the CPU and the `device` clause copy from the CPU to the GPU.

```
#pragma acc update host|device(list)
```

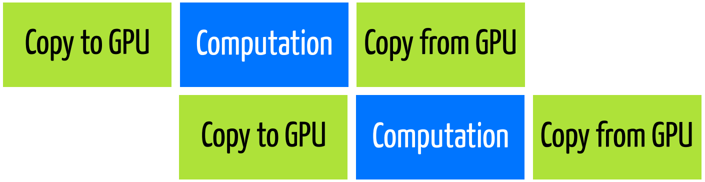
```
!$acc update host|device(list)
```

Serialized vs. Pipelined Execution

By breaking the computation of the Mandelbrot set in chunks, we do something called pipelining: breaking a large operation into smaller parts so that independent operations can overlap. By default OpenMP and OpenACC serialize the operations.



but we if want to overlap compution and memory copy operation.



Asynchronous Execution

In order to have a pipelined execution with OpenMP, we use a combination of the `nowait` and `depend` clauses. For the OpenACC version, we put the work in two asynchronous queues with the `async` clause.

```
#pragma omp target data \
map(alloc: image[0:WIDTH*HEIGHT])}
for(int block = 0; block < num_blocks; ++block) {
  int start = block * (HEIGHT/num_blocks);
  int end   = start + (HEIGHT/num_blocks);

  #pragma omp target teams distribute parallel for \
  collapse(2) depend(inout:image[block*block_size]) \
  nowait
  for (int y = start; y < end; y++) {
    for (int x = 0; x < WIDTH; x++) {
      image[x + y * WIDTH] = mandelbrot(x, y);
    }
  }

  #pragma omp target update \
  from(image[block*block_size:block_size]) \
  depend(inout:image[block*block_size]) nowait
}

#pragma omp taskwait
```

```
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; ++block) {
  int start = block * (HEIGHT/num_blocks);
  int end   = start + (HEIGHT/num_blocks);

  #pragma acc parallel loop collapse(2) \
  async(block%2)
  for (int y = start; y < end; y++) {
    for (int x = 0; x < WIDTH; x++) {
      image[x + y * WIDTH] = mandelbrot(x, y);
    }
  }

  #pragma acc update \
  self(image[block*block_size:block_size]) \
  async(block%2)
}

#pragma acc wait
```

Asynchronous Execution (OpenMP)

With OpenMP the `nowait` clause create a task detached to the main execution thread.
Dependence between the different task is defined with the `depend` clause.

```
#pragma omp target enter data map(alloc:A[:N], B[:N], C[:N])

#pragma omp target teams distribute nowait depend(out:A)
for (int i = 0; i < N; ++i) A[i] = i*4;

#pragma omp target teams distribute nowait depend(out:B)
for (int i = 0; i < N; ++i) B[i] = i*2;

#pragma omp target teams distribute nowait depend(in:A, B) depend(out:C)
for (int i = 0; i < N; ++i) C[i] = A[i] + B[i];

#pragma omp target exit data depend(in:C) map(from:C[:N]) map(delete:A[:N], B[:N])
```

Asynchronous Execution (OpenACC)

With OpenACC the **async** clause create a task detached to the main execution thread. You can specify the queue in which the task should be placed using an integer expression as argument to the clause. If no argument is specified, the task will be placed in the default execution queue.

```
#pragma acc enter data create(A[:N], B[:N], C[:N])

#pragma acc parallel loop async(1)
for (int i = 0; i < N; ++i) A[i] = i*4;

#pragma acc parallel loop async(2)
for (int i = 0; i < N; ++i) B[i] = i*2;

#pragma acc wait(1) async(2)

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

#pragma acc exit data copyout(C[:N]) delete(A[:N], B[:N]) wait(2)
```

Asynchronous Execution

Asynchronous execution can also be used in order to overlap the computation on the CPU and the GPU:

- to improve the throughput of your calculation
- to let the CPU a part of the calculation that might not be efficiently parallelized on a GPU

```
#pragma omp target teams distribute
            parallel for nowait
for (int i = 0; i < N; ++i)
  doSomethingGPU(i);

doSomethingCPU();

#pragma omp taskwait
```

```
#pragma acc parallel loop async
for (int i = 0; i < N; ++i)
  doSomethingGPU(i);

doSomethingCPU();

#pragma acc wait
```

Question?