Accelerating code: DIRECTIVES

Manuel Arenaz | June 6, 2019

Directive: *parallel*

- Directive that defines a parallel region, identifying the code region to be offloaded by the compiler.
- It starts parallel execution on the current accelerator.
- By itself of limited use. It needs to be combined with the work-sharing loop directive, which actually indicates to the compiler how to schedule the loop iterations on the accelerator.

```
#pragma acc parallel
  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
```

C and C++:

```
#pragma omp parallel [clause [[,] clause]…]
#pragma acc parallel [clause [[,] clause]…]
```

Fortran:

```
!$omp parallel [clause [[,] clause]…]
!$acc parallel [clause [[,] clause]…]
```

Directive: *kernels*

- Directive tells the compiler that a region ***may*** contain parallelism
- Relies on the automatic parallelization capabilities of the compiler to identify safe-to-parallelize code
- Compiler checks for data independence, then creates a separate kernel for each parallel

```
#pragma acc kernels
{
  for (i=0; i<N; i++)
  {
    y[i] = 0.0f;
    x[i] = (float)(i+1);
  }
  for (i=0; i<N; i++)
  {
    z[i] = 2.0f * x[i] + y[i];
  }
}
```

C and C++:

```
#pragma acc kernels [clause [[,] clause]…]
```

Fortran:

```
!$acc kernels [clause [[,] clause]…]
```

kernels or parallel?

- **Kernels:**
  - Only available in OpenACC (not in OpenMP)
  - Gives the compiler maximum leeway to parallelize and optimize
  - Relies most heavily on the compiler's ability to automatically parallelize the code
  - Performance varies by compiler

- **Parallel:**
  - Directive is an assertion by the programmer that it is both safe and desirable to parallelize the affected loop
  - Requires correct identification of parallelism in the code and removal of anything in the code that may be unsafe to parallelize
  - Incorrect assertion = incorrect results

Directive: *loop/for/do*

```
#pragma acc parallel loop
  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
```

- Directive for *work-sharing* in OpenACC
- By itself a parallel region is of limited use, but when paired with the loop directive the compiler will generate a parallel version of the loop for the accelerator.
- By using the loop directive the programmer asserts that the affected loop is safe to parallelize and allows the compiler to select how to schedule the loop iterations on the target accelerator.
- Clauses are used for correctness or performance

C and C++:

```
#pragma omp for [clause [[,] clause]…]
#pragma acc loop [clause [[,] clause]…]
```

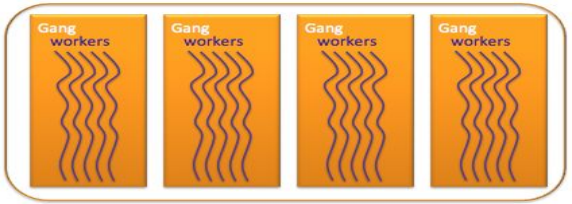
Fortran:

```
!$omp do [clause [[,] clause]…]
!$acc loop [clause [[,] clause]…]
```

The 3 levels of parallelism in the GPU

- Current accelerators and multicore CPUs support several levels of parallelism.
- The threads/tasks are grouped at different levels, but it is the compiler that is responsible for finding the best mapping for the hardware platform.

Level of parallelism	OpenACC	OpenMP 4.5
Coarse-grain: parallel execution across execution units	`gang`	`teams distribute`
Fine-grain: multiple threads of execution within a single execution unit	`worker`	`parallel for`
SIMD/vector: perform SIMD or vector operations within each execution unit	`vector`	`simd`



E.g. OpenACC on NVIDIA GPUs.

Directive: *atomic*

- Ensures only one thread can read/write (i.e. any form of access) a variable at any given time
- Use case: when one or more loop iterations need to access an element in memory at the same time data races can occur.
  - Use when a reduction is present, but the `reduction` operator cannot be used (e.g. for a sparse reduction)

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

C and C++:

```
#pragma omp atomic [clause [[,] clause]…]
#pragma acc atomic [clause [[,] clause]…]
```

Fortran:

```
!$omp atomic [clause [[,] clause]…]
!$acc atomic [clause [[,] clause]…]
```

Directive: *data / target data*

- **Data construct gives the programmer additional control over how and when data is created on and copied to or from the device.**
- **Clause: `copyin / map(to)`**
  - Copy the variable data to the device at the beginning of the region, and
  - release the space on the device when done without copying the data back to the host.
- **Clause: `copyout / map(from)`**
  - Create space for the listed variables on the device but do not initialize them.
  - At the end of the region, copy the results back to the host and release the space on the device.
- **Clause: `copy / map(tofrom)`**
  - Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region,
  - copy the results back to the host at the end of the region, and release the space on the device when done.

C and C++:

```
#pragma omp target data [clause [[,] clause]…]
#pragma acc data [clause [[,] clause]…]
```

Fortran:

```
!$omp target data [clause [[,] clause]…]
!$acc data [clause [[,] clause]…]
```

Shaping Arrays

- Help the compiler determine array size/shape (not typically necessary in Fortran)
- Helps the compiler ensure correct memory allocation on the device
- Add the shape specification to the data clauses, e.g.:

  `x[start:count]`

  `x[count]`

  where **`start`** is the first element to be copied and **`count`** is the number of elements to copy.
- Allows storing of only part of the array on the device

```
#pragma acc data create(x[0:N]) copyout(y[0:N])

!$acc data create(x(0:N)) copyout(y(0:N))
```

Directive: *routine / declare target*

- It tells the compiler to compile a given procedure for an accelerator as well as the host.
- Crucial to keep the existing code modularity
- Clause **seq**:
  - Specifies that the procedure does not contain nor does it call another procedure that contains a loop directive.
  - If other routines are called, then those routines must also be annotated with a routine directive.

```
#pragma acc routine seq
double foo() {...}

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

C and C++:

```
#pragma omp declare target [clause [[,] clause…]]
#pragma acc routine [clause [[,] clause…]]
```

Fortran:

```
!$omp declare target [clause [[,] clause…]]
!$acc routine [clause [[,] clause…]]
```

Best practice: `restrict` and `const`

- Use **`restrict`** in C whenever possible
  - Informs the compiler that the pointers are not aliased
  - This assists in the compiler in identifying which loops can be parallelized.
- Assist with read-only memory usage: use **`const`**
  - Informs the compiler that variable is a constant
  - Allows use of read-only memory for that variable if such a memory space exists on the accelerator

Both `restrict` and `const` are good programming practice in general, as it gives the compiler additional information that can be used when optimizing the code.

Walkthrough: Parallelizing HEAT

Goals

- Profile the code to identify hotspots
- Parallelize with kernels and measure performance
- Introduce restrict and const wherever possible
- Identify pattern of loops and parallelize each loop with parallel loop
- Compare performance with kernels and parallel loop