OpenACC. Part 2

Ned Nedialkov

McMaster University
Canada

CS/SE 4F03
March 2016

Outline

This presentation follows Chapter 15 of
David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Second Edition

parallel construct

- `parallel` specifies the block after it to be executed on the accelerator
- Gangs of workers are created to execute the parallel region
- The "gang leader" starts executing the parallel region
- Number of gangs and workers can be specified as e.g.

```
#pragma acc parallel num_gangs(1024)num_workers(32)
```

  This means $1024 \times 32 = 32,768$ workers
- # of gangs and # of workers is fixed during execution

Gang loop

The following parallel region will be executed redundantly by 1024 gang leads; each lead executes 2048 iterations

```
#pragma acc parallel num_gangs(1024)
{
   for (i=0; i<2048; i++)   {
    ...
    }
}
```

The following distributes the 2048 iterations to 1024 gangs, and each gang lead executes 2 iterations

```
#pragma acc parallel num_gangs(1024)
{
#pragma acc loop gang
   for (i=0; i<2048; i++)   {
    ...
    }
}
```

Worker loop

- A worker loop is a work-sharing construct
- The iterations are shared among the workers of a gang

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
#pragma acc loop gang
   for (i=0; i<2048; i++)    {
#pragma acc loop worker
      for (j=0; j<512; j++) {
        foo(i,j)
     }
   }
}
```

- Each gang executes 2 iterations of the “i” loop
- There are 32 workers per gang, executing 2×512 iterations of the “j” loop
- A worker executes $1024 \times 32 = 32$ instances of `foo`

Vector loop

Can express third level of parallelism or SIMD mode loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
    vector_length(32)
{
#pragma acc loop gang
   for (i=0; i<2048; i++)    {
#pragma acc loop worker
      for (j=0; j<512; j++) {
#pragma acc loop vector
          for (k=0; k<1024; k++) {
                foo(i,j,k);
           }
    }
  }
}
```

Possible mapping: gang → CUDA block, worker → CUDA warp, vector element → thread within warp
Mapping is not imposed by OpenACC
Compiler may choose different mapping

kernels construct

```
#pragma acc kernels
{
#pragma acc loop num_gangs(1024)
   for (i=0; i<2048; i++)
      a[i] = b[i];
#pragma acc loop num_gangs(512)
   for (i=0; i<2048; i++)
      c[i] = 2*a[i];
   for (i=0; i<2048; i++)
      d[i] = c[i];
}
```

- `kernel` specifies a kernel region
- may contain multiple kernel regions
- each may have different number of gangs, workers, and vector length
  they are specified in the loop constructs, not the kernels construct

kernels vs. parallel

- kernels: intention of the programmer, the compiler may or may not generate accelerator code
- loop construct: tells the compiler to generate such code only if the compiler generates kernels for these loops
- parallel: the compiler generates accelerator code

```
1  void foo(int *x, int *y, int n, int m)
2  {
3    int a[2048], b[2048];
4  #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
5    {
6  #pragma acc loop
7      // no data dependence
8      for (int i=0; i<2047; i++)
9        a[i] = b[i+1];
10 #pragma acc loop
11     // data dependence
12     for (int j=0; j<2047; j++)
13       a[j] = a[j+1]+1;
14 #pragma acc loop
15     // x and y may point to the same array
16     for (int k=0; k<2047; k++)
17       x[k] = y[k+1]+1;
18 #pragma acc loop
19     // no data dependence if n>=m
20     for (int l=0; l<m; l++)
21       x[l] = x[l+n]+1;
22   }
23 }
```

```
foo:
      4, Generating copy(x[:2048])
         Generating copy(y[:2048])
         Generating copy(a[:])
         Generating copy(b[:])
         Generating Tesla code
      8, Loop is parallelizable
         Accelerator kernel generated
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
              */
     12, Loop carried dependence of 'a' prevents parallelization
         Loop carried backward dependence of 'a' prevents vectorization
         Accelerator scalar kernel generated
     16, Complex loop carried dependence of 'y->' prevents
          parallelization
         Loop carried dependence of 'x->' prevents parallelization
         Loop carried backward dependence of 'x->' prevents vectorization
         Accelerator scalar kernel generated
     20, Loop carried dependence of 'x->' prevents parallelization
         Loop carried backward dependence of 'x->' prevents vectorization
         Accelerator scalar kernel generated
```

```
1  void foo(int *restrict x, int *restrict y, int n, int m)
2  {
3    int a[2048], b[2048];
4  #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
5    {
6  #pragma acc loop
7      // no data dependence
8      for (int i=0; i<2047; i++)
9        a[i] = b[i+1];
10 #pragma acc loop
11     // data dependence
12     for (int j=0; j<2047; j++)
13       a[j] = a[j+1]+1;
14 #pragma acc loop
15     // x and y cannot be the same due to restrict
16     for (int k=0; k<2047; k++)
17       x[k] = y[k+1]+1;
18 #pragma acc loop
19     // no data dependence if n>=m
20     for (int l=0; l<m; l++)
21       x[l] = x[l+n]+1;
22   }
23 }
```

```
foo:
      4, Generating copy(x[:2048])
         Generating copy(y[:2048])
         Generating copy(a[:])
         Generating copy(b[:])
         Generating Tesla code
      8, Loop is parallelizable
         Accelerator kernel generated
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
              */
     12, Loop carried dependence of 'a' prevents parallelization
         Loop carried backward dependence of 'a' prevents vectorization
         Accelerator scalar kernel generated
     16, Loop is parallelizable
         Accelerator kernel generated
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
             */
     20, Loop carried dependence of 'x->' prevents parallelization
         Loop carried backward dependence of 'x->' prevents vectorization
         Accelerator scalar kernel generated
```

```
1  void foo(int *restrict x, int *restrict y, int n, int m)
2  {
3    int a[2048], b[2048];
4  #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
5    {
6  #pragma acc loop
7      // no data dependence
8      for (int i=0; i<2047; i++)
9        a[i] = b[i+1];
10 #pragma acc loop
11     // data dependence
12     for (int j=0; j<2047; j++)
13       a[j] = a[j+1]+1;
14 #pragma acc loop
15     // x and y are not aliased, no dependence
16     for (int k=0; k<2047; k++)
17       x[k] = y[k+1]+1;
18 #pragma acc loop independent
19     // indpendent says the loop has no dependencies
20     for (int l=0; l<m; l++)
21       x[l] = x[l+n]+1;
22   }
23 }
```

```
foo:
      4, Generating copy(x[:2048])
         Generating copy(y[:2048])
         Generating copy(a[:])
         Generating copy(b[:])
         Generating Tesla code
      8, Loop is parallelizable
         Accelerator kernel generated
          8, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
              */
     12, Loop carried dependence of 'a' prevents parallelization
         Loop carried backward dependence of 'a' prevents vectorization
         Accelerator scalar kernel generated
     16, Loop is parallelizable
         Accelerator kernel generated
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
             */
     20, Loop is parallelizable
         Accelerator kernel generated
         20, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
             */
```

Data directives

- `copyin` copies from host to device
- `copyout` copies from device to host
- `copy` copies from host to device and back to host
- `create` creates a temporary on device
- ...

For more details, see OpenACC 2.0 http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf