OpenACC. Part II

Ned Nedialkov

McMaster University
Canada

October 2016

Outline

Some of 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 $1024/32 = 32,768$ workers
- # of gangs and # of workers are fixed during execution

Gang loop

Consider

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

- 1024 gang leads will execute this parallel region
- Each gang lead executes 2048 iterations
- Redundant executions!

Gang loop cont

Consider

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

- loop says share the work, or parallelize the loop that follows
- 2048 iterations are distributed to 1024 gangs
- Each gang lead executes 2 iterations

Worker loop

Consider

```
#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 outer loop
- Worker loop
  - iterations are shared among the workers of a gang
  - 32 workers per gang
  - execute $2 \times 512 = 1024$ iterations of the inner/worker 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];
}
```

- `kernels` tells the compiler “do the best you can do”
- may contain multiple kernels 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: more implicit, gives the compiler more freedom to parallelize
- parallel: the programmer specifies how to parallelize

Example

1 to 4 are identical in behaviour, but 5 is different

```
// 1
#pragma acc kernels loop
for( i = 0; i < n; ++i )
  a[i] = b[i] + c[i];

// 2
#pragma acc kernels
{
  for( i = 0; i < n; ++i )
    a[i] = b[i] + c[i];
}

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

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

// 5
#pragma acc parallel
{
  for( i = 0; i < n; ++i )
    a[i] = b[i] + c[i];
}
```

Example

```c
void foo(int *x, int *y, int n, int m)
{
  int a[2048], b[2048];
#pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
  {
#pragma acc loop
    for (int i=0; i<2047; i++)
      a[i] = b[i+1];
#pragma acc loop
    for (int j=0; j<2047; j++)
      a[j] = a[j+1]+1;
#pragma acc loop
    for (int k=0; k<2047; k++)
      x[k] = y[k+1]+1;
#pragma acc loop
    for (int l=0; l<m; l++)
      x[l] = x[l+n]+1;
  }
}
```

Which loops are parallelizable?

Example cont

```
void foo(int *x, int *y, int n, int m)
{
  int a[2048], b[2048];
#pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
  {
#pragma acc loop
    // no data dependence
    for (int i=0; i<2047; i++)
      a[i] = b[i+1];
#pragma acc loop
    // data dependence
    for (int j=0; j<2047; j++)
      a[j] = a[j+1]+1;
#pragma acc loop
    // x and y may point to the same array
    for (int k=0; k<2047; k++)
      x[k] = y[k+1]+1;
#pragma acc loop
    // no data dependence if n>=m
    for (int l=0; l<m; l++)
      x[l] = x[l+n]+1;
  }
}
```

```
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
```

```
void foo(int *restrict x, int *restrict y, int n, int m)
{
  int a[2048], b[2048];
#pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
  {
#pragma acc loop
    // no data dependence
    for (int i=0; i<2047; i++)
      a[i] = b[i+1];
#pragma acc loop
    // data dependence
    for (int j=0; j<2047; j++)
      a[j] = a[j+1]+1;
#pragma acc loop
    // x and y cannot be the same due to restrict
    for (int k=0; k<2047; k++)
      x[k] = y[k+1]+1;
#pragma acc loop
    // no data dependence if n>=m
    for (int l=0; l<m; l++)
      x[l] = x[l+n]+1;
  }
}
```

```
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
```

```
void foo(int *restrict x, int *restrict y, int n, int m)
{
  int a[2048], b[2048];
#pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
  {
#pragma acc loop
    // no data dependence
    for (int i=0; i<2047; i++)
      a[i] = b[i+1];
#pragma acc loop
    // data dependence
    for (int j=0; j<2047; j++)
      a[j] = a[j+1]+1;
#pragma acc loop
    // x and y are not aliased, no dependence
    for (int k=0; k<2047; k++)
      x[k] = y[k+1]+1;
#pragma acc loop independent
    // indpendent says the loop has no dependencies
    for (int l=0; l<m; l++)
      x[l] = x[l+n]+1;
  }
}
```

```
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
- ...

Summary from experience

- You have to have the right application to accelerate on a GPU
- Think about parallelism from the very beginning of your program development
- Parallelizing programs that have been written to run serially can be challenging; nontrivial restructuring is frequently needed to reveal parallelism
- Try to parallelize your program with OpenMP before moving to OpenACC
- If you cannot parallelize with OpenMP, practically no chances to get it working on a GPU
- Start with `kernels` and after you get your program working, experiment with `parallel`
- Let the compiler figure out number of gangs, workers, etc.
- Read, experiment, read, experiment ...