OpenACC

Ned Nedialkov

McMaster University

23 March 2023

Outline

parallel construct

gang loop

worker loop

vector loop

kernels construct

Data directives

Summary

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 a block 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 are fixed during execution

gang loop

```
#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!

```
#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

```
#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
- $2 \times 512 = 1024$ iterations of the inner/worker loop
- 32 workers per gang
- Each worker executes $1024/32 = 32$ 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);
    }
  }
}
```

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

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

```
1  void foo(int *x, int *y, int n, int m) {
2     int a[2048], b[2048];
3  #pragma acc kernels copy(x [0:2048], y [0:2048], a, b)
4     {
5  #pragma acc loop
6        for (int i = 0; i < 2047; i++)
7           a[i] = b[i + 1];
8  #pragma acc loop
9        for (int j = 0; j < 2047; j++)
10          a[j] = a[j + 1] + 1;
11 #pragma acc loop
12       for (int k = 0; k < 2047; k++)
13          x[k] = y[k + 1] + 1;
14 #pragma acc loop
15       for (int l = 0; l < m; l++)
16          x[l] = x[l + n] + 1;
17    }
18 }
```

Which loops are parallelizable?

Example cont

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

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

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.