Introduction to OpenACC

by
Srijith Rajamohan

Course Contents

This week:

- Introduction to OpenACC
- Data movement
- Express parallelism through directives
- Look at reduction examples
- Compute pixel values for a Mandelbrot set

2 / 52

What is OpenACC?

- Used for many-core architectures
- High-level programming model with emphasis on portability
- Emerged in 2011
- Use compiler directives to expose parallelism

3 / 52

Progressively accelerate the code

- Identify the parallelism
- Express parallelism through directives
- Express data locality
- Optimize

4 / 52

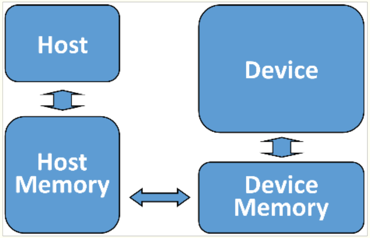
Advantages and Disadvantages

- High-level and platform independent
- Performance not as high as something like CUDA
- Cannot represent architecture-specific details without making it less portable
- Portability and performance are conflicting goals

5 / 52

OpenACC accelerator model

- Offload computation from host to accelerator
- Could be the same or different architecture, could be the same memory space or separate memory space



Abstract memory model

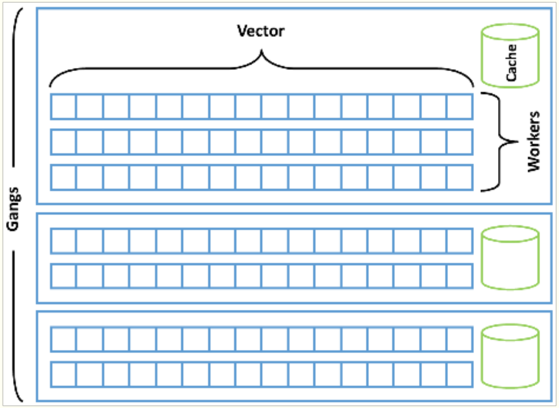
6 / 52

Parallelism

- gang: a threadblock, may or may not synchronize
- worker: warp (groups of 32 threads)
- vector: threads within a warp, operates on data that is a certain vector length

7 / 52

OpenACC accelerator model



Levels of parallelism

8 / 52

Directive syntax

Use the acc sentinel, which can be applied to code blocks.
Consists of directives and clauses.

Example

```
#pragma acc kernels
```

9 / 52

Porting cycle

- Parallelize loops: decorate loops with directives to provide the compiler with information to parallelize the loops
- Optimize data locality: it is necessary for the compiler to manage data movement between the host and the accelerator
- Optimize loops: provide additional information on how to restructure loops to expose parallelism or to reduce data movement

10 / 52

Compiling the code

Compile using the PGI compiler. Get compiler feedback using the ”-Mprof=ccff” flag

Example

```
pgcc -I../common -acc -ta=nvidia,time
-Minfo=accel,ccff -o laplace2d_acc laplace2d.c
```

11 / 52

Parallelize loops

- During this process, focus on moving as much of the computation to the accelerator as possible
- Ensure that the program gives correct results before focusing on data movement.

12 / 52

Kernels construct

- Identifies a region of code that may contain parallelism
- Relies on the compiler to automatically parallelize the code

13 / 52

Kernels construct

Example

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

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

Parallel loop

The parallel construct can be used along with the loop clause to parallelize the code across gangs.

Example

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

Parallel loop

- The programmer instructs that the loop is safe to parallelize
- Each loop needs to have the construct
- Sometimes the kernel cannot determine if a loop is parallelizable, e.g. when there is pointer aliasing or when two arrays share the same memory.

16 / 52

Loop - private

- The private clause specifies that each loop iteration requires it's own copy of the variable
- Loop iterators are private by default
- Any scalar accessed within a parallel loop is first private by default

17 / 52

Loop - reductions

- Reduction is similar to the private variables, except now a reduction is done at the end of the loop on all the private copies
- Result is returned in the variable after the loop exits
- Reduction may be specified on only a scalar variable

18 / 52

Jacobi iteration example

Example

```
while ( error > tol && iter < iter_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(inti=1;i<m-1;i++) {
  A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                 + Anew[j-1][i] + Anew[j+1][i]);
  error = fmax( error, fabs(A[j][i] - Anew[j][i]));
        }
}
}
...
```

19 / 52

Jacobi iteration example

Example

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

20 / 52

Jacobi iteration example

- The compiler can identify the inner loop based on the outer parallel construct
- Placing one on the inner loop tells the compiler that the loop is safe to parallelize
- Provide as much information to the compiler as possible

21 / 52

Jacobi iteration example - Kernels

Example

```
while ( error > tol && iter < iter_max ) {
error = 0.0;
#pragma acc kernels
for( int j = 1; j < n-1; j++) {
for(inti=1;i<m-1;i++) {
  A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                + Anew[j-1][i] + Anew[j+1][i]);
  error = fmax( error, fabs(A[j][i] - Anew[j][i]));
        }
}
}
...
}
```

22 / 52

Jacobi iteration example - Kernels

- The Kernels construct lets the compiler identify both the loops as parallelizable
- It also identifies the reduction operator
- If the Kernels construct was around the convergence loop, it would not have parallelized the loop
- Compared to the Parallel construct, you get more parallelism for less directives.

23 / 52

Performance comparison

- The Kernels construct does well, however the Parallel loop version does worse than the serial option
- This is because of the data transfer between the two parallel loops
- This is because the compiler performs the analysis on a region by region basis
- Data copied at the beginning and end of the region

24 / 52

Data locality

- Compilers must ensure that data is available if necessary
- The programmer will have knowledge of data needs that the compiler cannot determine
- Provide the compiler with as much information as possible to help prevent unnecessary data movement

25 / 52

Data Regions

- The Data construct facilitates the sharing of data between multiple parallel regions
- It must begin and end in the same scope
- Can be added around one or more Parallel regions

26 / 52

Data construct - Example

Example

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

27 / 52

Data construct

- The Data region in the example enables the arrays to be reused between the two parallel regions
- This removes the data copies between the two regions
- However, for optimal data movement you need to add data clauses

28 / 52

Data clauses

- copy - create space for the variables, copy data to the region at the beginning of the region and copy it back at the end
- copyin - create space for the variables on the device, copy the value at the beginning but don't copy it back to the host when done
- copyout - create space for the variables but do not copy the values in, copy the values back to the host when done
- present - the variables are already present, so no further action needs to be taken (used when a data region exists in a higher-level routine) ...

Data clauses continued...

- create - create space for the variables and release it at the end of the region, but do not copy to or from the device
- deviceptr - device memory that is managed outside of OpenACC, so these variables can be used without any address translation

30 / 52

Data construct - Example

Example

```
#pragma acc data pcreate(x[0:N]) pcopyout(y[0:N])
{
#pragma acc parallel loop
for (i=0; i<N; i++)
{
y[i] = 0.0f;
x[i] = (float)(i+1);
}

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

31 / 52

Data lifetimes

- Enter and exit data directives can be used to identify when data should be allocated or deallocated
- Precisely determine when data movement occurs
- They accept the create and copyin data clauses

Update construct

- Useful for synchronizing data between host and device memory
- This takes device and host clauses for copying to the device or to the host

33 / 52

Best practices tip

- Offload inefficient operations to move sections of the application to the device
- Doing this even if the code lacks sufficient parallelism is more efficient than transferring data back and forth
- For e.g. offload a serial section of the code with just 1 gang to the accelerator

34 / 52

Map parallelism to the hardware

- Gang clause
- Worker clause
- Vector clause
- Seq clause

35 / 52

Map parallelism to the hardware

- Apart from informing the compiler where to partition the loops, the programmer may also provide information about the number of gangs, workers and vector length to use for the loops.
- In the case of the Kernels directive, the gang, worker and vector clauses accept an integer parameter
- When using the Parallel directive, the information is presented on the Parallel directive itself as num_gangs, num_workers and vector_length clauses

36 / 52

Kernels - Example

Example

```
#pragma acc kernels
{
#pragma acc loop gang
for ( i=0; i<N; i++)
#pragma acc loop vector(128)
for ( j=0; j<M; j++)
...
}
```

37 / 52

Parallel - Example

Example

```
#pragma acc parallel loop gang vector_length(128)
for ( i=0; i<N; i++)
  #pragma acc loop vector
   for ( j=0; j<M; j++)
```

38 / 52

Device specific optimization

Example

```
#pragma acc parallel loop gang vector \
device_type(acc_device_nvidia) vector_length(128) \
device_type(acc_device_radeon) vector_length(256)
#pragma acc parallel loop gang vector_length(128)
for ( i=0; i<N; i++)
#pragma acc loop vector
for ( j=0; j<M; j++)
{
y[i] = 2.0f * x[i] + y[i]; }
```

39 / 52

Asynchronous operation

- Overlap the computation with the data transfer to minimize the performance penalty
- The async clause can be added to Parallel, Kernels and Update directives to specify that once the operation has been sent to the accelerator, the CPU may continue execution
- The wait directive instructs the runtime to wait for past asynchronous operations to complete before proceeding.

40 / 52

Asynchronous operation

Example

```
#pragma acc parallel loop async
{
c[i] = a[i] + b[i]
}
#pragma acc update self(c[0:N]) async
#pragma acc wait
```

41 / 52

Asynchronous operation - queues

- It would be useful to expose independent operations so that they could be executed independently
- Both async and wait have an integer number argument that specifies a queue number
- All operations placed in a certain execute in-order, but operations placed in different queues may operate in any order

42 / 52

Asynchronous operation - queues

Work is enqueued in queues 1 and 2 for the loops

Example

```
#pragma acc parallel loop async(1)
for (int i = 0; i<N; i++)
{
 a[i] = i; }
#pragma acc parallel loop async(2)
for (int i=0; i<N; i++)
{
b[i] = 2*i;
}
```

43 / 52

Asynchronous operation

wait(1) and async(2) ensures that work queue 2 does not proceed until queue 1 has completed.

Example

```
#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 update self(c[0:N]) async(2)
#pragma acc wait
```

44 / 52

Mandelbrot set

- Modify a simple application that generates a mandelbrot set
- Each pixel can be independently calculated, easy to parallelize
- Data transfer to copy the results back is expensive hence overlap it with computation

45 / 52

Mandelbrot set

Example

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

46 / 52

Step 1 - Computation

- Break the computation up into chunks of work that can be performed independently
- Determine the starting and ending bounds for each block

47 / 52

Step 1 - Computation

Example

```
int num_blocks = 8;
for(int block = 0; block < num_blocks; block++)
{
int ystart = block * (HEIGHT/num_blocks),
     yend   = ystart + (HEIGHT/num_blocks);
#pragma acc parallel loop
for(int y=ystart; y < yend ; y++)
{
  for(int x=0;x<WIDTH;x++) {
          image[y*WIDTH+x]=mandelbrot(x,y);
}
}
}
```

48 / 52

Step 2 - Data transfer

Example

```
int num_blocks = 8
int block_size = (HEIGHT/num_blocks)*WIDTH;
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ )
{
...

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

49 / 52

Step 3 - Overlapping computation

- The calculation and copying are still being done sequentially
- Make the operations asynchronous so that the copies and computation can happen simultaneously
- Use asynchronous work queues to ensure that the computation and data transfer within a single block are in the same queue
- The block number is convenient for this
- Now we need a wait directive after the block loop to ensure that all work completes before it is copied to the host

50 / 52

Step 3 - Overlapping computation

Example

```
int num_blocks = 8,
block_size = (HEIGHT/num_blocks)*WIDTH;
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ ) {
 ...
#pragma acc parallel loop async(block)
for(int y=ystart;y<yend;y++) {
 ...
#pragma acc update self(image[block*block_size:
block_size]) async(block)}
#pragma acc wait
```

Questions ?

52 / 52