OpenACC Lecture 3

Nick Johnson

EPCC
The University of Edinburgh
Scotland

Agenda

- Recap
- Other clauses
- More performance tuning
- Synchronisation
- Other Targets
- Questions from Today
- Rest of Today

Recap

From Practical 2 and the worked example, we saw that profiling can help us get good speed-up on a code.

We followed a logical work-flow by concentrating on a single *kernel* at first and then expanding outwards.

By keeping data on the device as much as possible, we can cut down a large bulk of the time.

By using a checksum, we can be reasonably certain that our code is running correctly.

Other clauses

In the codes we've looked at, there have been a few clauses that I've used without really discussing.
If you know OpenMP, these will be familiar to you:

- `reduction`
- `private`
- `firstprivate`

I also used some data movement clauses:

- `data present`
- `data create`

I'll now examine these in more detail.

Reduction

In an OpenACC (or OpenMP) reduction, the value of the reduction variable from each thread is combined at the end of the loop according to the reduction function.

Compare the two snippets:

```
#pragma acc parallel loop
for (i=0;i<100;i++){
  b += a[i];
}
```

```
#pragma acc parallel loop reduction(+:b)
for (i=0;i<100;i++){
  b += a[i];
}
```

In the left snippet, each gang will have a private copy of b which will accumulate it's values of a[i].
If we print this out on the host, it is unclear exactly which value will be used.

Reduction Gotchas

There is one problem with the previous example, two loop nests in a parallel region.

```
#pragma acc parallel
#pragma acc loop reduction(+:b)
for (i=0;i<100;i++){
  b += a[i];
}
#pragma acc loop
for (i=0;i<100;i++){
  a[i] = a[i] / b;
}
```

If we recall from Lecture 2, it is not clear that b will be correctly reduced by the time the second loop starts.
This could be a race condition.

Reduction Gotchas

There are two fixes for this problem:

```
#pragma acc parallel
#pragma acc loop reduction(+:b)
for (i=0;i<100;i++){
  b += a[i];
}
#pragma acc wait
#pragma acc loop
for (i=0;i<100;i++){
  a[i] = a[i] / b;
}
```

```
#pragma acc parallel loop reduction(+:b)
for (i=0;i<100;i++){
  b += a[i];
}
#pragma acc parallel loop
for (i=0;i<100;i++){
  a[i] = a[i] / b;
}
```

This won't happen in a kernels region OR if your compiler does the reduction on a worker or vector loop.

Note that the **PGI** compilers automatically insert the reduction clause in most cases. There is no way to turn this off!

Private

In OpenACC, scalars, reduction variables and loop iterators should be private by default.

There is no equivalent of OpenMP's `default(none)` in the current implementations BUT it is in v2 of the standard.

So, it is best to guide the compiler with other scalars by declaring them private.

If you are using a multi-level loop nest, with pointers & reductions, debugging a non-private variable can be tough.

Private

```
#pragma acc parallel loop
for (i=0;i<100;i++){
  for(j=0;j<100;j++){
    t += a[i*100+j];
  }
  c[i] = t;
}

#pragma acc parallel loop private(t)
for (i=0;i<100;i++){
  t = 0;
#pragma acc loop reduction(+:t)
  for(j=0;j<100;j++){
    t += a[i*100+j];
  }
  c[i] = t;
}
```

FirstPrivate

Like private, this is very similar to OpenMP. Except that support is still at bit hit-and-miss.

Here's the code in a contrived example:

```
#pragma acc loop firstprivate(b)
for (i=0;i<100;i++){
  b += a[i];
}
```

Every gang will start b with the value it had before the region.

Only use it if you really have to!

present & create

I used these in the Himeno worked example and you will have seen them in the code.

`create` - create space on the device for the arrays as you've specified on the host.

`present` - tell the compiler that you have already copied or created the data on the device.

```
#pragma acc data create(a)
{
<some code>
  sub_function(){
    #pragma acc data present(a)
  }
}
```

Data Ranges

The compiler is good at working out the sizes of arrays. Mostly.

When you use a dynamically allocated array, you might need to provide the data size.

You can also copy contiguous (and only contiguous) sub-arrays.

The syntax is `array[start:length]`.

```
int* restrict a = (int *)malloc(sizeof(int) * 1000);
#pragma acc data copy(a[10:20])
{
#pragma acc parallel loop
  for (i=10;i<30;i++){
    a[i] = a[i] + 10;
  }
} //end_data
```

Update

Sometimes, we want to move data from the host to the device (or vice versa) without doing a complete copy.

This is where we can use the `update` directive.

```
#pragma acc update device(a[10:10])
<some code that works on a[10] to a[19]>
#pragma acc update host(a[10:10])
```

It can be really useful to move intermediate results from a function back to the host.

Array decomposition

In previous examples we have used arrays with either a square or wide & shallow structure.

This might not map well to the accelerator hardware very well.

If we know the shape of our data, we might consider re-writing the loops to coalesce data better:

```
#pragma acc parallel loop
for (i=0;i<3;i++){
#pragma acc loop
  for(j=0;j<10000;j++){
    a[j] = b[i] + c[i] * d[i];
  }
}
```

```
#pragma acc parallel loop
for (i=0;i<3;i++){
#pragma acc loop
  for(j=0;j<10000;j++){
    a[i] = b[j] + c[j] * d[j];
  }
}
```

async & wait

These two clauses go hand in hand.

As we've discussed before, loops in parallel regions don't have an implicit barrier at the end of them.

BUT, parallel regions DO have a barrier at the end.

So, the host blocks until the parallel region has finished and cannot do anything else.

This can be wasteful so we use `async` to indicate that we don't have to block and `wait` to ask for an explicit barrier.

Both can take a single argument, an integer handle so you can have multiple sets of async & wait.

Example code

```
#pragma acc data create(a,b)
  {
    for(i=0;i<n;i++){
#pragma acc update device(a[i:0][0:n])
#pragma acc parallel loop
      for (j=0;j<n;j++){
        b[i][j] = a[i][j] * 2;
      }
#pragma acc update host(b[i:0][0:n])
    }
#pragma acc wait
  }
```

```
#pragma acc data create(a,b)
  {
    for(i=0;i<n;i++){
#pragma acc update device(a[i:0][0:n])
     async(i)
#pragma acc parallel loop async(i)
      for (j=0;j<n;j++){
        b[i][j] = a[i][j] * 2;
      }
#pragma acc update host(b[i:0][0:n])
     async(i)
    }
#pragma acc wait
  }
```

Other Targets

In this course we've only used the nVidia back-end via CUDA.

There are others available: AMD/ATI devices via OpenCL.

The OpenACC remains the same but performance will vary and almost certainly you will need to re-profile and perhaps apply some tweaks.

Present OR What?

You may have noticed that when you compile you get the statement:
`present_or_copyin`
This can be shortened to `pcopyin` and has the same meaning as `copyin` BUT will examine the device memory to see if the data is already present.

Runtime Schmuntime

You can do all sorts of fun things with the runtime calls, mostly related to getting information about the device.
Using the call `acc_set_device_num(int i, acc_device_t)` you could try using more than one card (hint - Erik has 2 GPUs per node).
You can also turn the cards on and off using `acc_init()` and `acc_shutdown()`.
There are other runtimes calls, but the standard explains them well.

CUDA Kernels

This is best illustrated with an example...:

```
#ifdef _OPENACC
#pragma acc data present(u[0:n1*n2*n3],v[0:n1*
   n2*n3],a[0:4],r[0:n1*n2*n3])
    {
#pragma acc host_data use_device(u,v,r,a)
      {
        resid_cuda(u,v,r,&n1,&n2,&n3,a);
      }
    }
#else /* _OPENACC */
#error
#endif /*_OPENACC */
```

Remaining Time

What's coming up...

- Coffee (we have some time so take it easy).
- Practical 3 - I throw you in at the deep end!
- Q&A / Finish Practicals / Work on your own codes.

More Help?

One of the best places to look for help is in the pgroup.com forums. You don't need to be a PGI customer to access them but obviously, they only answer questions using the PGI compiler.

You can also email me : Nick.Johnsoned.ac.uk