OpenACC Lecture 2

Nick Johnson

EPCC
The University of Edinburgh
Scotland

Agenda

Recap & Follow-up

Gotchas

Regions, again.
- Parallel
- Kernels
- Scheduling

Bugs & Features
- Pointers
- IEEE754

Profiling

You have now all compiled your first six OpenACC codes!
Welcome to the club.

Rest of the morning:

- Another lecture
- A worked example

Scheduling

You should have seen lines like the following when compiling:

```
   18, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
```

This is the clever part. The compiler has taken the loop to be accelerated and worked out how to map it to the target hardware available.
Here it has applied gang level and vector parallelism, with the vector size being 256.
Being a multiple of 256 it makes sense for a CUDA-based target.

Scheduling

- Scheduling is simply the mapping of the loop nests the hardware available.
- Compilers are different and conservative! You will usually get different results for different compilers for anything but the simplest codes.
- In general, I advise letting the compiler do the hard work unless you are certain you can do better (ie you know the codes very well).
- We'll return to looking at how to configure this later...

Regions

In the first lecture and practical, I glossed over this important part. Now it's time to look in more detail at regions and loops.
There are **two** types of regions in the OpenACC model. It's slightly non-sensical but is an artifact of the proprietary predecessor directives.

- Parallel: where you, the programmer manage the parallelism.
- Kernels: where the compiler tries to do it for you.

Let's look in each in more detail.

Parallel Regions

Back to our old example...

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

What's actually happening here?

- I've split the line into it's two formal parts, `acc parallel` and `acc loop`.
- `acc parallel` tells the compiler to start a number of **gangs** which redundantly execute everything inside them until they hit...
- `acc loop` which tells the compiler to share the work between the gangs.
- This is very, very similar to the work sharing model in OpenMP which shouldn't be surprising as there is significant overlap between the two groups.

More on parallel...

- There is no gaurantee that if you have two loops inside a parallel region, the first will complete before the second starts.
- It's like an OpenMP for loop with the nowait directive.
- That's why I used separate parallel regions for each nest in Practical 1.
- You can add `#pragma acc wait` after each loop nest if you wish to make sure everything is synced.

```
#pragma acc parallel
#pragma acc loop
for (i=0;i<100;i++){
  a[i] = b[i];
}
#pragma acc loop
for (i=0;i<100;i++){
  c[i] = b[i]*a[i]*2;
}
```

What's happening here?

- The compiler is entirely at liberty to map some of the first nest to some of the gangs and some of the second nest to the rest!
- This could be a race condition, there is no gaurantee that a[10] is dealt with by the same thread in each nest.
- Compiler might get it right. It might not. For more complex codes, it could be horrendous.

Kernels Regions

Back to our old example...

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

What's actually happening here?

- I've split the line into it's two formal parts, `acc kernels` and `acc loop`.
- `acc kernels` tells the compiler to analyse the following loop nest(s), convert each to a kernel and lauch on the accelerator, in order!
- In this mode, the compiler uses auto-parallelisation routines to work out what to do with your loops and how to schedule them on the accelerator.

More on kernels...

- There is a gaurantee that if you have two loops inside a kernels region, the first will complete before the second starts.
- For simple codes, this is a quick way to get going.
- The parallelism and scheduling is implicit and the compiler takes care of it for you.
- There is one gotcha, pointers. We'll cover this later.

Each loop nest in the kernels construct is compiled and launched separately. In CUDA terms, each loop nest becomes a separate CUDA kernel. In particular, this means that the code for the first loop nest will complete before the second loop nest begins execution.

– Michael Wolfe, PGI.

Scheduling

There are three levels of parallelism expressable in OpenACC, `gang`, `worker`, `vector`.
They can be placed on loops to help guide the compiler.
Consider the below example of our single loop where we guide the compiler.

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

This wont make much difference and we are best to let the compiler do it.

Scheduling

Here's the output from my machine:

```
main:
     17, Accelerator kernel generated
         18, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
     17, Generating present_or_copyout(a[0:])
         Generating present_or_copyin(b[0:])
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
```

It's exactly the same as before.

Scheduling

What about a doubly nested loop?

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

```
main:
     17, Accelerator kernel generated
         18, #pragma acc loop gang /* blockIdx.x */
         21, #pragma acc loop vector(256) /* threadIdx.x */
     17, Generating present_or_copyout(a[0:])
         Generating present_or_copyin(b[0:])
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     21, Loop is parallelizable

Runtime = 55us
```

Scheduling

What about a doubly nested loop?

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

```
main:
     17, Accelerator kernel generated
         21, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
     17, Generating present_or_copyout(a[0:])
         Generating present_or_copyin(b[0:])
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     18, Loop is parallelizable

Runtime = 110us
```

Vectorisation

The mapping of the loops to the hardware can seem a bit like magic.
Lets look at the mapping in a CUDA context since we are using nVidia hardware:

- `gang` - maps to SM's.
- `worker` - maps to threadblocks.
- `vector` - maps to threads / warps.
- `seq` - execute the loop sequentially.

In the first example, the compiler mapped the parallelism of our loop to both gangs (SMs) and vectors (threads). In CUDA, we would always try to get coallesced memory accesses by having each thread operate on adjacent elements of an array. This greatly improves speed. Each gang executed one (or more) of the i-loops and within an i-loop, the j-loops were mapped across threads.
In the second example, we asked the compiler to vectorise across the i-loop and map the j-loops across gangs. It chose to split the loops across gangs and vectors but this wasn't so efficient given the doubling in runtimes.

There are two things that might catch you out when using OpenACC:

- Pointers.
- Maths.

Let's take a (brief) look at these and how to avoid them.

In C, using dynamically allocated memory is generally the norm. As we saw in Practical 1, we can use it with OpenACC without too many problems. However:

- With a dynamically allocated array, the compiler cannot be certain you are not aliasing or even what the size of the array is. This makes it hard to decide on vector lengths and parallel decomposition.
- The compiler will often throw a warning and refuse to parallelise your loops for you.
- You can take this in two ways:
  - Use the restrict keyword in your declaration:
    `int * restrict a = (int*)malloc...`
  - Use the independent keyword in the `kernels loop` line

Using the previous single loop example with a size of 10000, using either trick forces the compiler to sequentialise the loop and give me a runtime of 932us. Using either trick got it down to 70us!

If you run comparisons with host code or use host compute checksums (which you should do), you might notice that for bigger summations, the values diverge. This is normal and expected as the GPU might engage in some tricks to get a faster runtime at the expense of strict IEEE754 compliant maths.

We could spend a whole day discussing the merits or otherwise of this in scientific computing but it would send us all to sleep!

There are two methods to deal with the problem using the PGI compiler:

- Request IEEE754 compliant maths by using the compiler flag `-Kieee`.
- Introduce a tolerance into your algorithm of 10^{-14}.

Remember also that you won't get byte similar results from two different CPUs or possibly even the same CPU on different days. If you can improve your result by running a few more iterations, that's the best option.

Profiling

So far, we have used quite simplistic tools to measure the performance of our codes. For simple codes, this is okay but as we scale up, it becomes more useful to use more tools.

There are three or four tools that I typically use for this type of work.

- `printf` and OpenMP timers - portable and mostly every system has OpenMP these days.
- The cuda profile - can be hard to interpret.
- The compiler runtime - varies by compiler, can be hard to understand.
- nvprof & nVidia Visual Profiler - can be useful to interpret cuda profile.

nvprof

Provided by nVidia as part of the CUDA SDK and available on Erik.

Can be simple to use and the output is easily readable.

Execute your code as normal but prefixed by `nvprof`

```
[njohnso1@fermi1 ~]$ nvprof ./demoacc4
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   46.63   10.82us       1   10.82us   10.82us   10.82us  [CUDA memcpy DtoH]
   43.26   10.04us       1   10.04us   10.04us   10.04us  [CUDA memcpy HtoD]
   10.10    2.34us       1    2.34us    2.34us    2.34us  main_17_gpu
```

CUDA Profile

A raw, verbose version of what you get from nvprof.
Gives you slightly more information wrt CPU and GPU time.

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla C2050
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6894980df08
method,gputime,cputime,occupancy
method=[ memcpyHtoDasync ] gputime=[ 9.408 ] cputime=[ 10.000 ]
method=[ main_17_gpu ] gputime=[ 4.192 ] cputime=[ 16.000 ] occupancy=[ 0.667 ]
method=[ memcpyDtoHasync ] gputime=[ 8.064 ] cputime=[ 6.000 ]
```

PGI Runtime

This is what we've used already in Practical 1.
Enabled by exporting PGI_ACC_TIME=1 in your terminal (or in submit.bash).

```
Accelerator Kernel Timing data
/home/h019/njohnso1/demoacc4.c
  main  NVIDIA  devicenum=0
        time(us): 49
        16: data copyin reached 1 times
             device time(us): total=16 max=16 min=16 avg=16
        17: kernel launched 1 times
            grid: [79]  block: [128]
             device time(us): total=19 max=19 min=19 avg=19
            elapsed time(us): total=87 max=87 min=87 avg=87
        21: data copyout reached 1 times
             device time(us): total=14 max=14 min=14 avg=14
```

Performance

There is an obvious question:

Why not write that kernel in CUDA/OpenCL for speed?

Generally, we could, **BUT** it breaks portability.

The rule-of-thumb is that if we get within 80% of CUDA/OpenCL speed then we are happy.

In many cases we get faster code because hand tweaking CUDA/OpenCL can be time consuming and prone to errors.