# OpenACC Lecture 1

Nick Johnson

EPCC
The University of Edinburgh
Scotland

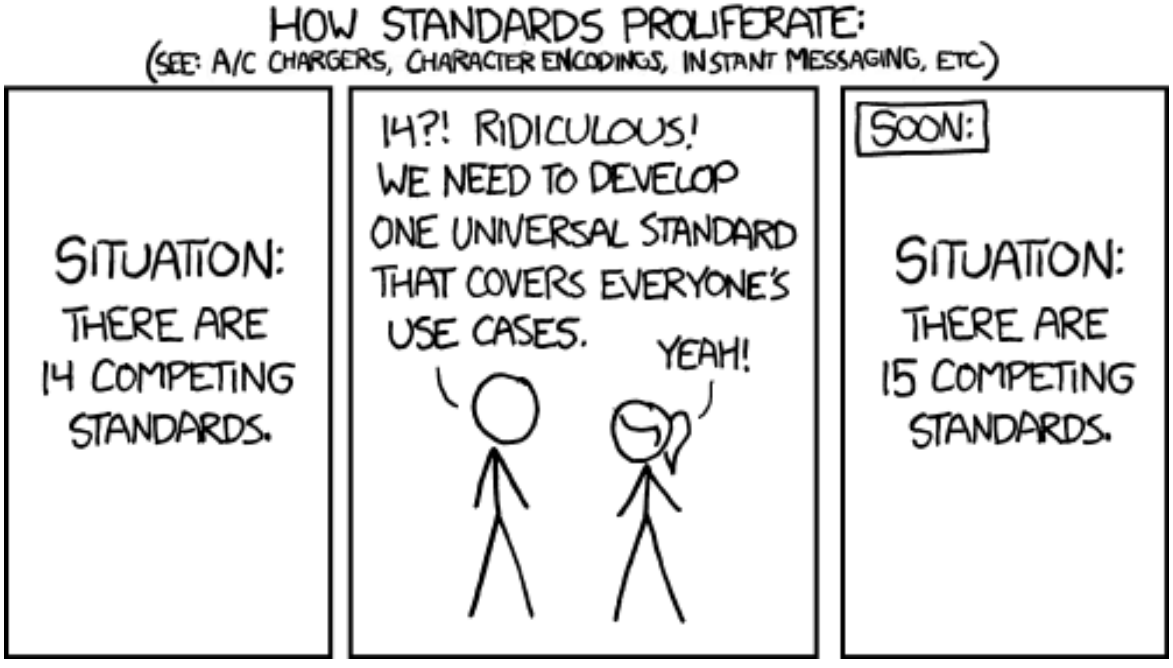
# Agenda

# Directives for GPUs

- Around 2010, compiler vendors started to look at directives for GPUs.
- Everyone developed their own set of directives.
- Moved some of the burden from the programmer to the compiler.
- Mostly targetting nVidia as the *target*.
- Not intended to be soley for GPUs but it has mostly ended up that way.

# An (OpenACC) standard is born

*The nice thing about standards is that you have so many to choose from*
Andrew S. Tanenbaum
Computer Networks, 2nd ed., p. 254




xkcd.com/927

- Rather than wait for OpenMP to incorporate accelerators, nVidia, CAPS, Cray & PGI create the OpenACC board.
- The board release v1 of the OpenACC standard at SC 11.
- Initial support came fairly quickly after that.
- Performance quickly began approaching that of vendors' own directives.
- OpenMP v4 RC2 has support for accelerators.

# Where are we today?

- CAPS, Cray and PGI all have compilers available to buy that support OpenACC.
- You can get a free compiler from ULL if you are very nice to Ruyman Reyes.
- With some small exceptions, code is portable between the three.
- Some compiler still favour particular methods.
- This is improving and for many operations, performance varies little between the compilers.

# Meet the new board

- OpenACC board has now expanded (as of SC12)
- New members: TU Dresden, U Houston, ORNL, Allinea, CSCS
- New supporters: Sandia NL, RogueWave, Georgia Tech

# Why not just use CUDA/OpenCL?

- It's Hard!
- Not very high level.
- I am a rubbish coder and make mistakes.
- Compiler writers are clever.

# The path to Exascale

- Most likely Exascale systems will be hetergenous.
- Quite possibly with different accelerators / off-load devices available.
- Programming those systems yourself might take a long time & be hard (even for HPC people).
- Using a directives-based language is one good way forward, let clever (IS)Vs do the work for you.

# Execution Model

In detail:

- Host-directed execution with attached GPU accelerator
- Main program executes on "host" (i.e. CPU)
- Compute intensive regions offloaded to the accelerator device under control of the host.
- "device" (i.e. GPU) executes parallel regions typically contain "kernels" (i.e. work-sharing loops), or kernels regions, containing one or more loops which are executed as kernels.
- Host must orchestrate the execution by: allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the parallel region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory.
- Host can usually queue a sequence of operations to be executed on the device, one after the other.

# Memory Model

In detail:

- Memory spaces on the host and device distinct
- Different locations, different address space
- Data movement performed by host using runtime library calls that explicitly move data between the separate
- GPUs have a weak memory model
- No synchronisation between different execution units (SMs)
- Unless explicit memory barrier
- Can write OpenACC kernels with race conditions
- Giving inconsistent execution results
- Compiler will catch most errors, but not all (no user-managed barriers)

OpenACC:

- data movement between the memories implicit:
  managed by the compiler, based on directives from the programmer.
- Device memory caches are managed by the compiler with hints from the programmer in the form of directives.

OpenACC *code* takes two forms:

- Directives
  - Typically used to indicate data movement
  - Code to be accelerated
  - Mapping of parallelism
  - Most common *code* we will be using today
- Runtime Calls
  - Typically used to setup and shutdown device
  - Can be used to configure device
  - Can give some added functionality but must be used carefully.
  - Compilers often support calls from the standard & their own proprietary calls
  - **Be Careful!**

OpenACC works for the three main HPC languages. Here's the syntax:

- C/C++ `#pragma acc`
- Fortran `!$acc ... !$acc end`

The rest of the slides use the C syntax but the practicals are mostly in C and Fortran.

# Regions

- The way to express parallelism in OpenACC is to mark up a loop to be accelerated in a region.
- There are two different methods to do this: `Parallel` & `Kernels`.
- The differences are quite subtle and often open to debate.
- I'll use `parallel` here and go in to the details in the next lecture.

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

# Regions

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

This instructs the compiler to do two things (three really, but more on that later)

- Turn the `for` loop into a kernel to run on the target device.
- Use its own logic to decide on the parallel decomposition depending on the target.
- Copy that kernel to the device.
- Run the kernel.
- Deal with any output.

# Data

We know that the device has a completely separate memory and therefore we need to be careful about where the data we are operating on is.

OpenACC has two modes of dealing with data:

- Implicit Data Regions: where we let the compiler decide what to do
- Explicit Data Regions: where we tell the compiler what to do

No method is foolproof. It is entirely possible for the compiler to get it wrong or for us to lie to the compiler and for it not to notice.

# Implicit Data Regions

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

Consider our previous simple example:

- This is an implicit data region: We didn't tell the compiler to do anything with the data.
- So long as the compiler can determine the size of `a` and `b`, it can copy them to the device along with the kernel.
- Scalars wuch as `i` and constants are copied for you.
- If you use `a` later on in the code, it will be copied back.
- Compiler feedback will tell you about this.

# Explicit Data Regions

Consider our previous simple example, with one modification:

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

- We added the `#pragma acc data` line with some braces.
- This defines a data region.
- Data copied on to the device within this region will persist with the region.
- We've not been clever about it, compiler still has to work out sizes, and work out what to copy.

# Explicit Data Regions

Consider our previous simple example, with one modification:

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

- We added the `copyin` clause.
- The compiler will now copy the arrays `a` and `b` to the device.
- Compiler still has to work out sizes.
- Could use `a` or `a` later inside the data region for another kernel without having to copy in again.

# Explicit Data Regions

Other data clauses

- `copyout`: Create an array on the device and copy to the host at the end of the region (ie, **YOU** will initialize the array).
- `create`: Create an array on the device (good for temporary arrays).
- `present`: The data is already on the device (used in function calls).
- `copy`: Copy data to and from the device. Use sparingly! You generally only need to copy *big data* to the device and a small result back.

# Compiling a code

There is a bewildering number of options to use with any compiler which can give you any result you wish. Correct or otherwise.
To begin with, we will keep things simple and concentrate on the pgcc compiler from Portland Group:

```
$ pgcc -acc -Minfo=accel mycode.c -o mycode
```

This is very much like compiling a normal C code, except we have used the `-acc` flag to enable OpenACC support.

# Complete code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <openacc.h>

int main(int argc, char *argv[])
{
  int a[100];
  int b[100];
  int i;

  for(i=0;i<100;i++){
    b[i] = i;
  }

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

  return 0;

}
```

# Compilers

Disclaimer!

I will the the PGI compiler for these examples because it is the one available on the service machine for today (Erik). Other compilers are avaialble.

# Compiler output

What happens when we run our previous exemplar code through the compiler?

```
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
```

# Running output

What happens when we run our exemplar?

```
$ ./mycode
$
```

Not surprising, we didn't ask for any output!

# Runtime output

The runtime can gives some good information but requires some decoding.
Enable the output and see the result:

```
$ export PGI_ACC_TIME=1
$ ./mycode

Accelerator Kernel Timing data
/home/h019/njohnso1/demoacc.c
  main  NVIDIA  devicenum=0
        time(us): 48
        17: data copyin reached 1 times
             device time(us): total=16 max=16 min=16 avg=16
        17: kernel launched 1 times
            grid: [1]  block: [256]
             device time(us): total=21 max=21 min=21 avg=21
            elapsed time(us): total=117 max=117 min=117 avg=117
        22: data copyout reached 1 times
             device time(us): total=11 max=11 min=11 avg=11

$
```

# Your turn

Now it's your turn...

- Take a copy of the codes.
- There are three versions of the code for both static and dynamically allocated memory.
- Try to get them compiling and running (makefile available).
- Look at the output from the compiler and try to understand what's going on.
- There will be things we haven't covered (yet) so ask lots of questions.
- The helpers and I will help you out.
- Coffee is after the practical so if you want to take an earlier break, please do so.
- The code is free so feel free to send it to others, play with it, break it, adapt it.