Lecture 11

DEBUGGING OPENCL

Debugging OpenCL

- Parallel programs can be challenging to debug
- Luckily there are some tools to help
- Firstly, if your device can run OpenCL 1.2, you can printf straight from the kernel.

```
__kernel void func(void)
{
  int i = get_global_id(0);
  printf(" %d\n ", i);
}
```

- Here, each work-item will print to stdout
- Note: there is some buffering between the device and the output, but will be flushed by calling clFinish (or equivalent)

Debugging OpenCL 1.1

- Top tip:
  - Write data to a global buffer from within the kernel

    ```
    result[ get_global_id(0) ] = … ;
    ```

  - Copy back to the host and print out from there or debug as a normal serial application
- Works with any OpenCL device and platform

Debugging OpenCL – more tips

- **Check your error messages!**
  - If you enable Exceptions in C++ as we have here, make sure you print out the errors.
- Don't forget, use the err_code.c from the tutorial to print out errors as strings (instead of numbers), or check in the cl.h file in the include directory of your OpenCL provider for error messages
- Check your work-group sizes and indexing

Debugging OpenCL - GDB

- Can also use GDB to debug your programs on the CPU
  - This will also leverage the memory system
    - Might catch illegal memory dereferences more accurately
  - But it does behave differently to accelerator devices so bugs may show up in different ways
- As with debugging, compile your C or C++ programs with the `–g` flag

Debugging OpenCL - GDB

- Require platform specific instructions depending on if you are using the AMD® or Intel® OpenCL platform
  - This is in part due to the ICD (Installable Client Driver) ensuring that the correct OpenCL runtime is loaded for the chosen platform
  - Also different kernel compile flags are accepted/required by different OpenCL implementations
- Remember: your CPU may be listed under each platform – ensure you choose the right debugging method for the *platform*

Using GDB with AMD®

- Ensure you select the CPU device from the AMD® platform
- Must use the –g flag and turn off all optimizations when building the kernels:

```
program.build(" -g -O0" )
```

- The symbolic name of a kernel function "**__kernel void foo(args)**" is "__OpenCL_foo_kernel"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

```
break __OpenCL_foo_kernel
```

- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code
- AMD® recommend setting the environment variable `CPU_MAX_COMPUTE_UNITS=1` to ensure deterministic kernel behaviour

Using GDB with Intel®

- Ensure you select the CPU device from the Intel® platform
- Must use the –g flag and specify the kernel source file when building the kernels:

```
program.build(" -g -s
/full/path/to/kernel.cl" )
```

- The symbolic name of a kernel function "**__kernel void foo(args)**" is "foo"
  - To set a breakpoint on kernel entry enter at the GDB prompt:

```
break foo
```

- Note: the debug symbol for the kernel will not show up until the kernel has been built by your host code

Debugging OpenCL – Using GDB

- Use ***n*** to move to the next line of execution
- Use ***s*** to step into the function
- If you reach a segmentation fault, ***backtrace*** lists the previous few execution frames
  - Type ***frame 5*** to examine the 5th frame
- Use ***print varname*** to output the current value of a variable

Oclgrind

- A SPIR interpreter and OpenCL simulator
- Developed at the University of Bristol
- Runs OpenCL kernels in a simulated environment to catch various bugs:
  - `oclgrind ./application`
  - Invalid memory accesses
  - Data-races (`--data-races`)
  - Work-group divergence
  - Runtime API errors (`--check-api`)
- Also has a GDB-style interactive debugger
  - `oclgrind -i ./application`
- More information on the Oclgrind Website

GPUVerify

- A useful tool for detecting data-races in OpenCL programs
- Developed at Imperial College as part of the CARP project
- Uses static analysis to try to prove that kernels are free from races
- Can also detect issues with work-group divergence
- More information on the GPUVerify Website

```
gpuverify --local_size=64,64 --num_groups=256,256 kernel.cl
```

Other debugging tools

- AMD® CodeXL
  - For AMD® APUs, CPUs and GPUs
    - Graphical Profiler and Debugger
- NVIDIA® Nsight™ Development Platform
  - For NVIDIA® GPUs
    - IDE, including Profiler and Debugger
- GPUVerify
  - Formal analysis of kernels
  - http://multicore.doc.ic.ac.uk/tools/GPUVerify/

Note: Debugging OpenCL is still changing rapidly - your mileage may vary when using GDB and these tools