Advanced Multicore Programming

Dr Andrew Moss

[1]BTH Karlskrona

September 11, 2011

Performance Models

Two approaches to performance

- “Theoretical” model : a.k.a what nVidia and AMD tell us should happen
- Practical model : a.k.a what can we measure

Practical considerations

- Cannot run timing code on the GPU
- Cannot execute any direct measurements
- Cannot even observe what happens directly
- Hardware $\Rightarrow$ Driver $\Rightarrow$ API $\Rightarrow$ Your Code

Measuring Performance

The GPU operates asynchronously

- Entirely different clock domain
- DMA
- Asynchronous message transfers to/from host

Impossible to just measure a sequence of commands

- Must measure sequences of commands and then receiving a response
- At least one RTT in the measurement
- Each step in the transfer will add some overhead
- We cannot measure small things precisely

Measuring Performance II

Standard assumption

If we measure a large set of things then the overhead will be amortised

- Basically wrong . . .
- But it also doesn't really matter . . .
- Really we want to know if sequence A is faster than sequence B
- We can assume the overhead is roughly constant
  - Probably is if the memory traffic in A is the same as B. . .

Basic approach

- Take a time sample on the CPU
- Queue up a sequence of commands on the GPU
- Execute them
- Wait for a synchronisation (clFinish(), or synchronous memory transfer. . .)
- Take a second sample

Measuring Performance III

The numbers that we get are wrong

- Overhead millions of times larger than the measurement
- But, it's hard to do better
- As long as we try to measure large things we will be fine

Taking time samples on the CPU

```
#define FCLOCK(a,b) __asm__ __volatile__( \
"xorl %%eax, %%eax;\n\
cpuid;\n \
rdtsc" : "=a" (a), "=d" (b) : : "ebx", "ecx" );
```

- Numbers returned are machine dependent (clock cycles)
- Can translate them using `/proc/cpuinfo`

Measuring Performance IV

Using the macro is straight-forward

- Make sure you do 64-bit math, 2^{32} cycles is not very long...

Using FCLOCK()

```
Uint64 start;
Uint32 lo,high;
FLOCK(lo,hi);
start = ((Uint64)hi<<32ULL) + (Uint64)lo;
```

- The 32-bit values are necessary (ABI defined to RDTSC)
- The combination is a bit tricky (compilers make this awkward)
- Take two sample, the difference is a time measurement
- Very high precision; compared to the GPU execution we can treat this as exact

Synchronisation

We need a synchronisation point before we measure

- Stops the measurement happening before the work

Three kinds of interesting synchronisation

- Wait events, internal to the GPU
- Memory transfers, synchronisation for both sides
- Completion of a command queue

Typical Workload

1. Send initial data to the GPU
2. Process data locally on the GPU (repeats)
3. Retreive solution from the GPU

Synchronisation II

One possible approach

- SAMPLE
- Send
- Process
- Receive (synchronous transfer)
- SAMPLE

Definitely an upper bound on the duration

- Processing could not have overlapped with the send.
- The second sample could not have overlapped with the receive.

Probably quite inaccurate.

- Two memory transfers included in the sample.
- Card/Host transfers are slow.

Synchronisation III

One possible approach

- Send (synchronous transfer)
- SAMPLE
- Process
- Receive (synchronous transfer)
- SAMPLE

Definitely still an upper bound on the duration

- Processing definitely starts after we start the clock
- The second sample could not have overlapped with the receive.

Probably still inaccurate.

- Memory transfer still included in the sample.
- Card/Host transfers are slow.

Synchronisation IV

One possible approach

- Send (synchronous transfer)
- SAMPLE
- Process
- clFinish()
- SAMPLE
- Receive

Definitely still an upper bound on the duration

- Processing definitely starts after we start the clock
- Execution must have finished before the second sample

Probably still inaccurate.

- Different time domains, DMA, context switching etc.
- As good as we will get for raw measurements though. . .

Timing Approaches

First batch of exercises used the supplied framework

- Implicit synchronisation barrier with the vsync.
- The geometry update kernel includes locks with the OpenGL memory.
- The OpenGL double buffering is locked to the vsync.

Very Coarse Measurement (binary answer)

Will these kernels execute in $\frac{1}{\text{fps}} - \epsilon$ seconds?

- Where ϵ is some transfer overhead running in sequence with the OpenCL code.

Both approaches have their place

- The more accurate sampling is when you want to know how a kernel scales (e.g Assignment 1).
- The coarse sampling is useful when you want to know if your technique will hit the time budget for a particular framerate.

Event Queues

Each compute context uses a command queue

- Invocations of a kernel are commands in this queue
- clSetKernelArg to change the arguments of the kernel function
- This call is stateful
- Use clEnqueueNDRangeKernel to place a kernel execution in the queue with the current args
- No guarantees about the ordering used between multiple kernel executions
- The range used determine how many copies of the kernel, which IDs they use

Examples in the demo code; more information in the man-pages / OpenCL reference

Event Queues II

Simplest examples only have a single kernel in the command queue

- Once you write more complex code you will need multiple passes
- Changing arguments are essential to altering behaviour
  - Switching memory buffer sources and targets
  - Passing parameters like “current pass” to change operation
- The grid ID may not always give you enough information
  - Naive matrix multiplication is $O(n^3)$ . . .
  - One way to split it up is tiling the inner product
  - Split one of the three parameters into ranges
  - Could use a 3D grid to express this, or a constant parameter
- Depends on which way is easier to express your problem
  - i.e the best approach will depend on the problem

Event Queues III

Sometime you want finer grained control over ordering

- If your first kernel reads buffer A and write to buffer B...
- You have a second kernel that operates on B then updates A...
- Controlling their order is pretty important
  - Standard concurrency issues like generating garbage

Every kernel execution in OpenCL can generate a synchronisation event

- Each kernel can wait on a set of events before it executes
- This should be quite low weight...
  - No guarantees, but these primitives should be implemented in the GPU, not the driver

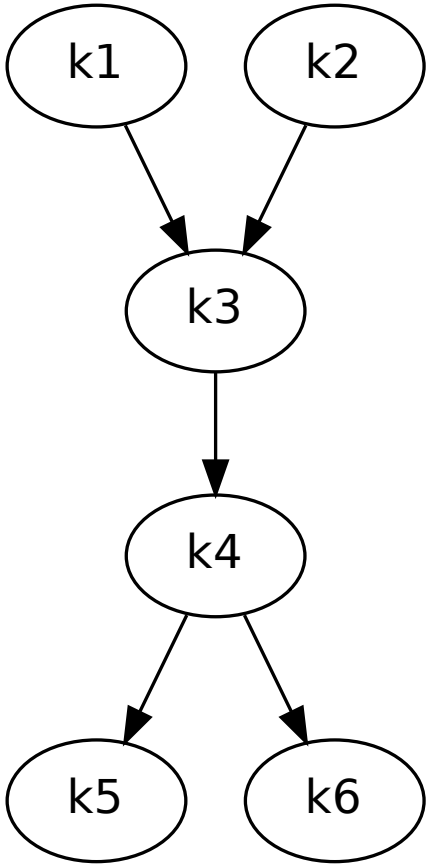
Event Queues IV

Each kernel can produce at most one event

- Each kernel can wait on an arbitrary set of events
- Allows you to build a DAG of kernel executions
- Synchronisation between layers in the DAG...
  - Kernels within a layer have an arbitrary order

If you have multiple kernels on your final layer

- clFinish



Finer Grain Control

Fences / Barriers etc

- TBC

Exercises

You have a bunch of simple kernels over particles

- Or you should have, make sure you save them to different source files this time. . .

Implement profiling code, using both the coarse and finer timing methods

- Find out how the kernels that you've implemented scale to larger numbers of particles