Ellie Makin (erm67)

Computer Design Supervision 4

Additional questions

1. **With load linked / store conditional, why does the store alter the source register containing the value to write?**

So that we have a way to know if the store went through or not. Another indicator could be used, but the source register is the most natural. If we didn't have a way to check if the `sc` had gone through, then we would not be able to make most of the useful synchronisation primitives.

2. **Assuming a single instruction, xchg, that can do an atomic exchange of a register and memory location, how would you implement a naive spin lock?**

```
lock:
    mv t0, 0
    xchg t0, 0(a0)
    beqz t0, lock     #check if lock was already zero

unlock:
    mv t0, 1
    sw t0, 0(a0)
```

3. **What does a memory barrier do?**

Optimised processors may not always have the property that memory accesses from one core are seen in program order by other cores (e.g. RISC-V doesn't guarantee this). A memory barrier is an instruction (`fence` in RISC-V) which guarantees that all memory (or device) accesses before the barrier finish before any of the memory accesses after the barrier begin.

4. **Why do we place a memory barrier after taking a lock and before releasing the lock, but not the other way round (i.e. before taking and after releasing)?**

If we place a barrier before taking a lock, but not after, it means that memory accesses that occur after the lock is taken may be seen to occur before the lock has been taken properly, which violates the assumptions of the lock and may have side effects. Similarly, placing a barrier after an unlock instead of before, may mean that the unlock appears to happen before the code in the critical section before the it. putting the barrier on the wrong side in either of these cases is no better than leaving it out, since there are no restrictions on the code either side of the critical section.

5. **Describe SIMT, one way GPUs exploit parallelism. Apart from performance, what are the other benefits of SIMT execution?**

Single instruction, multiple threads. The threads in question are GPU threads, which means that many different GPU cores will each be running one of these threads at once. These cores are grouped into collections of usually about 32, called a warp. A warp of threads will all execute the same instructions at once, using predicate registers to handle any conditional execution. Each core in a warp will store multiple paused threads, so that if a high latency operation needs to take place, it can switch to one of the paused threads while it waits. Code intended to be run by the GPU should have minimal conditional branching, so that the maximum number of cores are executing instructions at the same time.

The idea behind SIMT is to exploit the benefits of many simple cores, rather than a few complex cores, to allow for maximum throughput of instructions, and limit the overhead of instruction fetch. This allows the cores to be quite small, so many of them can be put on the same chip. However, the extra state needed to store the paused threads may make this less true, despite the performance boost.

6. **How does the processor pipeline differ from a general-purpose CPU to allow SIMT execution?**

Conditional execution is no longer handled by changing the `pc`, and instead the predicate registers are set and unset in order to dictate which cores should be executing each instruction. The instruction fetch and decode stages are only executed once for each warp, which means that it should probably occur outside of the cores.

7. **What is the basic way GPUs allow branching within the different threads?**

The process is called masking. If a predicate register stack is used, then whenever an `if` statement (or instruction equivalent) is executed, the current state of the predicate register is pushed into the stack, and all of the threads for which the predicate holds have their predicate registers set to `true`. Within the body of the `if` statement, only the threads where the predicate register is `true` will execute instructions (including changes to the predicate register). If an `else` statement is reached, then the predicate register is set in the inverse way to the preceding `if` statement (accounting for nested ifs). When the end of the `if` block is reached, the predicate register is popped from the stack.

a. **What does this imply about the way you should write code for GPUs?**

Code should be written to minimise conditional execution - especially nested conditionals - or at least minimise the number of threads for which the predicate is false. Similarly, while loops which may loop for longer on some threads can be treated as an if statement which is nested so many levels deep, and if looping is required (if it’s even possible?), then a for loop in which each thread is always executing is best.

b. **Why does it make sense to do as little work as possible within the target of an infrequently-taken branch?**

This will maximise the average number of threads executing at once. Each instruction where a thread is idle is a wasted cycle, so if many threads are idle for many instructions, that’s a lot of wasted cycles.

8. **Why do GPUs rely on parallelism between warps as well as SIMT? What bottleneck does this target?**

If all the warps were executing synchronously, then if just one thread stalled, all the others would have to either pause or do a context switch, which would be very inefficient for a large number of threads. Separating them into small-but-not-too-small groups avoids this problem, while still reducing the overhead from instruction fetch.

9. **Compare and contrast the OpenCL programming model with CUDA.**

Both OpenCL and CUDA use a programming model which can be used to specify GPUs. However, OpenCL uses a very abstract model of the device architecture it runs on, which means that it can also be run by other kinds of processor, is well as GPUs from other vendors than Nvidia.

10. **Why is energy efficiency the “new fundamental limiter of processor performance”, as Borkar and Chien say?**

As Moore’s law slows down, we run into a new problem, which is that we can fit many transistors onto a single chip, but we cannot power them all. There are also improvements to be made with respect to reducing overheads on a microarchitectural scale.

11. **What types of application domain might benefit from approximate computing?**

Applications that don’t require 100% accuracy, such as graphics, audio, sensor analysis. Generally, these are applications that deal with real-world data, or human interaction, rather than scientific or mathematical requirements.

2019 Paper 5 Question 3

a. **Describe each of the four models defined by OpenCL’s specification.**

- Platform model - a platform consists of a host with one or more devices that it may interact with, such as a CPU and several GPUs within a machine.
- Execution model - Each device has a command queue, which allows the host to send commands to each device. The command queue contains event objects, which are wrapper objects for commands, that specify a dependency ordering between them.
- Kernel Programming model - Kernels are the programs or functions which run on the devices. They are broken into work-items, which represent a single thread running of this function. work-items are grouped into work-groups, which are like multiprocessors. Kernels are compiled at runtime, in order to account for variety in device architecture.
- Memory model - OpenCL specifies an abstract memory model, with various degrees of locality. Global, which is accessible by all kernels, constant, which is like global but simultaneous read-only, local, which is accessible to all items in a work group, and private, which is private to each work item.

b. **Describe the different types of memory available to OpenCL kernels.**

See above. There is also host memory, which is accessible to systems outside of OpenCL.

c. **Contrast how calls to a kernel, e.g. DAXPY, are invoked and grouped for execution in OpenCL compared with CUDA.**

In OpenCL, there is a much more general model of the underlying system than in CUDA, which means that much more specification and setup must be done before running a kernel (with the benefit that it is highly platform independent). The kernel is compiled at runtime onto the specific device, which makes startup time longer. In OpenCL, the number of work items is an n-dimensional range, as opposed to CUDA, which has 2 dimension parameters, for the number of grids and the number of thread blocks.

d. **Describe, with the aid of a diagram, how a GPU executes data-parallel kernels efficiently, including the two main pieces of hardware support.**

I don't fully understand what this question is asking.

e. **Describe the trade-offs between using a GPU or a specialised accelerator for tasks containing data-level parallelism.**

Or this one.