

ICA_Supervision_4_Answers_Zhou_Kyra_Micha

[Question 1\(y2015p5q3\)](#)

[Question 2\(y2016p5q3\)](#)

[Question 3\(y2019p5q3\)](#)

[Question 4](#)

[Question 5](#)

[Question 6](#)

[Question 7](#)

[Question 8](#)

[Question 9](#)

[Question 10](#)

[Question 11](#)

[Question 12](#)

[Question 13](#)

[Question 14](#)

[Question 15](#)

[Question 16](#)

[Question 17](#)

[Question 18](#)

Question 1(y2015p5q3)

(b) (i) What are the semantics of load linked and store conditional instructions?
[4 marks]

(ii) Describe the synchronisation method that the following code performs by adding comments to it.

```
membar
label1: ll    r2, 0(r1)
        sub   r2, r2, #1
        sc    r2, 0(r1)
        beqz  r2, label1
label2: load  r2, 0(r1)
        bneq  r2, label2
```

[4 marks]

- (i):
 - ll (load-link) operation would load the value stored at register r1 into register r4; before the sc (store-conditional) operation, we can modify the value in r4, whatever we like.
 - When sc is called, it checks whether the value in register r1 has changed since the last ll, if not, then overwrite the value at r1 with the value from register r4, otherwise abort the store operation.

```
lock: mov r3, #1 // 1 means lock is taken, so load 1 into r3
      ll r4, 0(r1) // linked load: load the value of r1 into r4
      sc r3, 0(r1) // store conditional: check whether the value of r1 is unchanged
      beqz r3, lock // if sc failed: loop on failure
      bneq r4, lock // check whether lock is taken: if r4 = 1, then someone has locked, r4 = 0 -> no one has locked it
```

- (ii):

```
membar // memory barrier ensure all memory operations before would be executed before starting executing the label1 loop
label1: ll r2, 0(r1) // load-link would store and link value from r1 into r2
        sub r2, r2, #1
        sc r2, 0(r1) // store conditional: synchronise the value at r1 if r1 has not been modified since last load-link operation (r2 = 1); otherwise set r2 = 0 to indicate a failure
        beqz r2, label1 // loop on failure
```

```
label2: load r2, 0(r1)
        bneq r2, label2 // if r2 = 1(lock has been taken by someone), then loop on failure, else if r2 = 0(no one has taken the
lock), then continue executing
```



Comments:

Question 2(y2016p5q3)

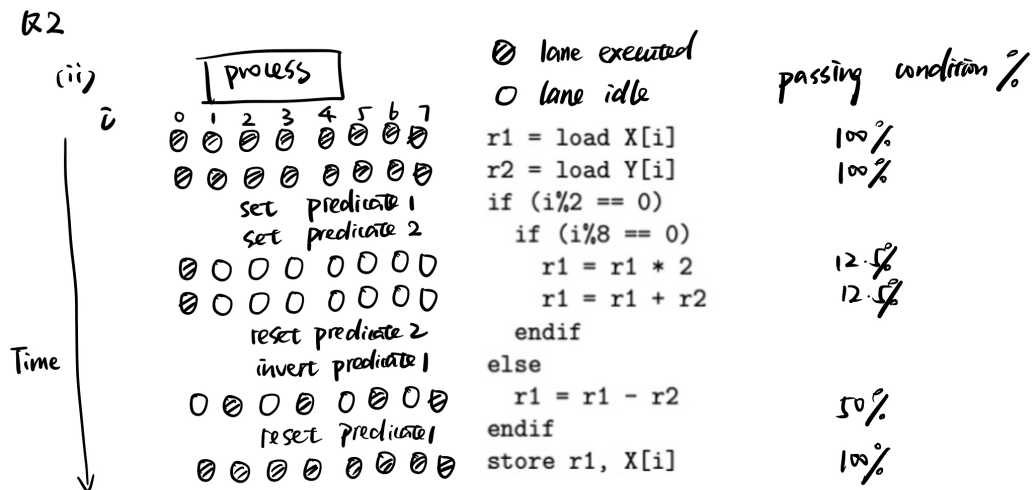
- (b) Consider the following pseudo-code that is run on a SIMD processor with 8 lanes, where i gives the lane number.

```
r1 = load X[i]
r2 = load Y[i]
if (i%2 == 0)
    if (i%8 == 0)
        r1 = r1 * 2
        r1 = r1 + r2
    endif
else
    r1 = r1 - r2
endif
store r1, X[i]
```

- (i) Describe how the processor can support branch divergence between the different lanes. [4 marks]

- The processor uses conditional execution, which is implemented by several predicate registers. The predicate registers would hold conditions.
- During execution, all threads(lanes) execute the same instructions; when we encounter the `if` statement, the predicate is set on, and lanes are executed if the condition holds; otherwise, the lanes would be idle.
- When we later enter the `else` statement, the predicate registers are inverted, so the lanes that are idle previously would execute at this stage.
- After exiting the if-else statements, the predicates are reset.

- (ii) With the aid of a diagram, show the utilisation of the SIMD lanes for each pseudo-code operation, hence calculate the code's efficiency (overall utilisation of the SIMD lanes). [6 marks]



Efficiency = $E(X)$ where X is the passing condition fractions
 † nodes at each epoch

$$= \frac{100\% \times 3 + 12.5\% \times 2 + 50\%}{6}$$

$$= 62.5\%$$

(iii) What architectural technique do GPUs employ to allow them to perform useful work even though the loads from X and Y often cause stalls?

[2 marks]

- Using the multithreading structure, where we have much more thread states than processing resources, we can have multiple warps, each with a certain amount of threads.
- Warp parallelisation helps hide the latency from memory operations because we can switch between warps at computation time. If a stall occurs in one warp, then we can call in instruction on another warp to hide the latency of the previous warp by doing useful work in another warp.
- We use the warp scheduler to choose which warp to execute for latency hiding



Comments:

Question 3(y2019p5q3)

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

[4 marks]

- GPU over specialised accelerator:
 - GPU does not depend on the CPU, so a CPU could have less load and work on other important tasks simultaneously.
 - GPU has its own memory, so it will not take memory from the CPU, where the specialised accelerator on the CPU would use a vast amount of memory.
- Specialised accelerator over GPU:
 - GPU is more expensive than an integrated specialised accelerator on the CPU

- GPUs often have an overheating problem because they consume more power; we need to add more fans to cool down GPUs.



Comments:

Question 4

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

- In the following example, `r2` is the source register containing the value to write into `r1`, storing the original value.
- In the store conditional operation, the store alters the source register and uses it as a flag or indication of success/failure of the store conditional operation. If `r1` hasn't been modified since the last load linked operation, we need to indicate that the store conditional operation was successfully executed, thus storing `1` into `r2`. If `r1` has been changed since the last load linked operation, we need to loop on failure, so set `0` into `r2`.

```
ll r2, 0(r1)
sub r2, r2, #1
sc r2, 0(r1)
```



Comments:

Question 5

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?

```
spin_lock:
    mov r1, #1 // 1 means lock is taken, so load 1 into r1
    xchg r1, [locked] // atomic exchange of a register and memory location
    bneq r1, spin // r1 stores the value of memory location [locked], if r1 = 0, then lock hasn't been taken; if r1 = 1, then lock has been taken, so loop on failure
    ret // locked the memory address, and return

spin_unlock:
    addi r1, zero, 0 // reset register
    xchg r1, [locked] // atomic exchange of a register and memory location
    ret // unlocked the memory address, and return
```



Comments:

Question 6

What does a memory barrier do?

- A class of instructions where memory operations (e.g., read/writes) occur in the order the programmer specified. It is a hardware concept that can implement mutexes and semaphores.
- Modern CPUs optimisations lead to out-of-order execution of memory operations under a multi-thread environment; this out-of-order execution behaviour may lead to unpredictable results.



Comments:

Question 7

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)?

- By placing a memory barrier after taking a lock and before releasing the lock, we can ensure the memory operations for the lock comes before entering the critical section and before memory operations for releasing the lock.
- If we place a memory barrier before taking the lock and after releasing the lock, then the order of memory operations in the locking and unlocking phases is not preserved; they might be out-of-order, thus leading to unpredicted behaviours under a multi-threaded environment.

```
// lock
// membar
// cs
// membar
// unlock

// membar
// lock
// cs
// unlock
// membar
```



Comments:

Question 8

Explain the relaxed consistency model. What are the alternatives?

- Load and store operations to different addresses can bypass each other in a relaxed consistency model, although relative ordering between operations to the same addresses is maintained. One of the consequences of out-of-order memory operations is that impossible results might be introduced.
- An alternative model is the sequential consistency model, which provides an illusion of sequential consistency. All reads and writes by a single processor are seen in the order they occur. A sequential consistency model on top of relaxed consistency processors is preferred for multi-threaded applications accessing shared data.



Comments:

Question 9

Describe SIMT; one way GPUs exploit parallelism.

- Single instruction multiple threads: Threads are grouped together for efficiency, with only one instruction for a warp (a group of threads).
- Each processor runs many threads concurrently; each thread has its own registers and memory; all threads execute the exact instructions on different data items; hence, it helps achieve data-level parallelism.




Comments:

Question 10

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

- GPU is designed for throughput; it has multiple cores, each core is relatively simple and runs many threads concurrently.


- Each thread has its own registers and memory; threads are grouped together as a warp, all threads in the same warp execute the same instructions but on different data items to enable SIMT execution.
- In CPU sequential execution, each instruction works on scalar data, and only one value is read from or written to memory at a time. But in GPU, since we have multiple threads(with their own registers and ALUs), we can execute the same instruction in all the threads concurrently, thus improving the throughput and enabling SIMT execution.

 Comments:

Question 11

What are the implications for other parts of the processor for providing SIMT (e.g. the data cache)?

- Each thread has its own private local memory to store stack frames and spilling registers. We can add a data cache for the private memory to improve memory access efficiency.
- Each multi-threaded processor would have shared memory for threads, enabling communication between threads with high bandwidth. We could add a data cache to store the shared data among threads within the same multi-threaded processor.

 Comments:

Question 12

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

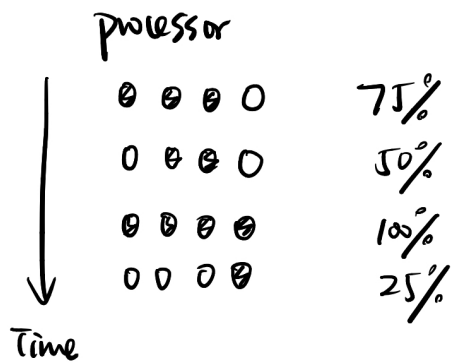
- Using conditional execution with a predicate register that holds conditions, we would set the predicates to allow one branch execution, then invert it to enable another branch to execution, and eventually reset predicates once the branching statements exist.
- All threads still execute the same instructions, but threads only execute if the condition holds at that timestep; otherwise, the thread sits idle, and no computation is performed, which leads to inefficiency.

 Comments:

Question 13

How do we calculate the efficiency of code on GPU? Sketch an example.

- Efficiency = $E(X)$, the expected value/mean of X , where X is the fractional of codes that pass the condition at each epoch.



$$\text{Efficiency} = E(x) = \frac{75\% + 50\% + 100\% + 25\%}{4}$$

$$= 62.5\%$$



Comments:

Question 14

Explain warp parallelism and SIMT parallelism.

- Warp parallelism is used to hide the latency introduced by the multithreading structure.
 - On each processor, we have multiple warps that would execute independently. Inside each warp, we have multiple threads that could execute concurrently and have their own registers, ALUs, and private local memory.
 - If stalls occur in one warp, then call an instruction on another warp to hide the latency of the previous warp. The selection of warp is decided by the warp scheduler, based on when operands are ready.
- SIMT parallelism is a way to achieve data level by grouping threads together as one warp that could execute the same instruction concurrently on different data items, thus increasing throughput.
 - Each processor runs many threads concurrently; each thread has its own register, ALUs and private local memory; all threads execute the same instructions but on different data items at the same time.



Comments:

Question 15

What are the types of memory available within a GPU, and what are their relative advantages or disadvantages?

- Types:
 - Private local memory, per thread
 - Shared memory, per multi-threaded processor
 - Global GPU memory across GPU
 - Constant and texture memories
- Relative Pros & Cons:

- Private local memory, Global GPU memory, constant and texture memories are in external DRAM to be large and cached.
- Shared memory per processor is located within each multi-threaded processor, with higher bandwidth and low latency. Hence it could be used for communication between threads.
- Constant and texture memories can be cached in specialised caches (e.g., constant cache broadcasts scalar values), improving graphics processing performance.
- Global memory is also available to the host processor, enabling communications between the host processor and the GPU.
- Private local memory per thread is used for storing stack frames so that registers can execute the same instruction with different data items concurrently.



Comments:

Question 16

Summarise the main message from lecture 13 in 1-3 sentences?

- The load-link and store-conditional operations are used to implement the spinlocks, which are used to synchronise primitives.
- Memory barriers are used to ensure the ordering of the operations before and after the memory barrier instructions.



Comments:

Question 17

Summarise the main message from lecture 14 in 1-3 sentences?

- SIMT, which stands for single instruction multiple threads, is a way to achieve data-level parallelism for GPU.
- The branching problem in SIMT is solved by conditional execution using predicate registers.
- Multithreading hides latency with warp parallelism. Each processor would have multiple independent warps, with each warp a group of threads that execute the same instruction on different data simultaneously.



Comments:

Question 18

Summarise the main message from lecture 15: GPU memory in 1 sentence?

- The memory hierarchy is implemented to compromise the trade-off between latency and physical limits of the memory size and balance isolation and sharing.
- Caches are used to reduce bandwidth for a multithreading structure, although warp parallelism has to be applied to hide memory latency already.



Comments: