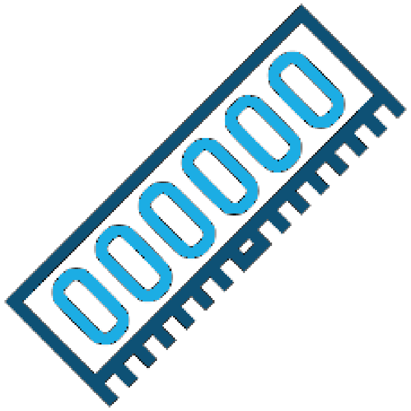
# CS3210

## Parallel Computing

Changes from Monday in Green



## Tut 3

Mon (4pm)
Tues (2pm)

## Admin Updates

- Assignment 2 released last weekend - due 19 Oct, 11am
  - Topic: CUDA parallelisation of hash-based proof-of-work
  - Weightage: 10%
  - Please start on this early - there are much fewer reserved Compute Cluster nodes than lab machines
- Assignment 1 still being graded (no ETA)
  - Late submissions for Part 2 are still accepted
- End-tutorial Quiz 3 later

## Review
## Memory Coherence (I)

- Cache coherence problem arises on multi-processor systems where each processor has a private cache
  - Multiple copies of the same data exists in different caches
  - A write by one processor to a location may not become visible → other processors may keep reading stale data

- Why not unify all private caches into a shared cache?
  - Increased latency from larger cache size
  - Central bandwidth bottleneck and negative interference (cache conflicts/thrashing)

## Review
## Memory Coherence (II)

- Goal: reading a memory location should return the last value written by any processor

- Three properties
  - **Program order**: preserves sequential order of each processor
  - **Write propagation**: writes by one processor must become visible to all others
  - **Write serialization**: all writes to a location must be seen in the same order by all processors

Review

## Coherence and Consistency

- Coherence: defines requirements for observed behavior of reads and writes to the same memory location
  - All processors must agree on order of reads/writes to X

- Consistency: defines behavior (constraints) of reads and writes to different memory locations, as observed by other processors
  - Coherence only guarantees that writes to address X will *eventually* propagate to SMs
  - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Review

## Memory Operation Orderings

- A program defines a sequence of loads and stores: a program order

- Four types of memory operation orderings
  - $W \rightarrow R$: write to X commits before subsequent read from Y
  - $R \rightarrow R$: read from X commits before subsequent read from Y
  - $R \rightarrow W$: read to X commits before subsequent write to Y
  - $W \rightarrow W$: write to X commits before subsequent write to Y

Review

## Consistency Models (I)

- Sequential consistency: preserves all four orderings
  - Effect of each memory operation must be visible to all processors prior to next operation on any processor

- Relaxed consistency: allow relaxation of ordering of memory operations in absence of data dependencies
  - Data dependency arises when one operation writes to X, which is then read by a later operation
  - Goal: hide memory latencies to improve performance

Review

## Consistency Models (II)

- Summary table (weak consistency models not covered)

| Property | SC | RC: TSO | RC: PC | RC: PSO |
|---|---|---|---|---|
| Preserves data dep? | Yes | Yes | Yes | Yes |
| Preserve $W \rightarrow R$ | Yes | No | No | No |
| Preserve $R \rightarrow R$ | Yes | Yes | Yes | Yes |
| Preserve $R \rightarrow W$ | Yes | Yes | Yes | Yes |
| Preserve $W \rightarrow W$ | Yes | Yes | Yes | No |
| Write propagation | Yes | Yes | No | Yes |

**Return value of any write (even from other processor) before write is propagated or serialised**

Q1

# Memory Consistency

- Consider this shared memory program on 3 processors

**P1**

```
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A
```

**P2**

```
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X
```

**P3**

```
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

- Goal of this program: ensure **`print A`** and **`print B`** print same value
  - All variables are initialized to **`0`**

## Q1(a)
# Memory Consistency

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X

        P3
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

- Under the sequential consistency model, what are the possible values of $A$, $B$, $T$, $X$ after the program finishes?
  - Note this is different from the values printed by P1 and P3!
- What is preserved under sequential consistency?
  - All four types of memory orderings
  - Write propagation and serialization (implied)

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X

        P3
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

Q1(a)

## Memory Consistency

- What can be inferred from this instruction dependency graph?

$$\begin{tikzcd}
1 \arrow[d] & 5 \arrow[d] & 10 \arrow[dl] \arrow[d] \\
2 \arrow[ur] \arrow[dd] & 6 \arrow[d] & 11 \arrow[dd] \\
 & 7 \arrow[dl] \arrow[d] \arrow[dr] & \\
3 \arrow[d] & 8 \arrow[d] & 12 \arrow[d] \\
4 & 9 & 13
\end{tikzcd}$$

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X

        P3
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

Q1(a)

## Memory Consistency

- Answer

| A | B | T | X | Possible execution scenario |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 10, 11, 1, 2, 5 - 9, 3, 4, 12, 13 |
| 1 | 1 | 1 | 3 | 1, 2, 10, 5 - 9, 3, 4, 11, 12, 13 |
| 1 | 3 | 1 | 3 | 1, 2, 10, 5 - 8, 3, 4, 11, 9, 12, 13 |
| 3 | 3 | 1 | 3 | 1, 2, 10, 5 - 7, 11, 8, 9, 3, 4, 12, 13 |
| 3 | 3 | 3 | 3 | 1, 2, 10, 11, 5 - 9, 3, 4, 12, 13 |

- Prove the following are impossible:

| A | B | T | X | Reason |
|---|---|---|---|---|
| 1 | 3 | 1 | 1 | B = 3 implies final value of X = 3 |
| 3 | 1 | 3 | 1 | A = 3 implies B = 3 by prog. order |

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X

        P3
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

## Q1(b) Memory Consistency

- The code fails its goal even under sequential consistency - give one such execution scenario
  - General idea: ensure $(11)$ gets executed after $(8)$ but before $(9)$

- Sample execution
  - $(1) \rightarrow (2) \rightarrow (10) \rightarrow (5) \rightarrow (6) \rightarrow (7) \rightarrow (8) \rightarrow (3) \rightarrow (4)\ [A = 1] \rightarrow (11) \rightarrow (9) \rightarrow (12) \rightarrow (13)\ [B = 3]$

## Q1(c)
# Memory Consistency

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 7] T = X
[ 8] A = X
[ 9] B = X

        P3
[10] Z = 2
[11] X = 3
[12] while (T == 0);
[13] print B
```

- Rewrite the code such that the code achieves its goal under sequential consistency
  - ➢ General idea: prevent (11) from executing in between (8) and (9)

- Answer
  - ➢ Swap (10) and (11) so that X = 3 is set first before Z = 2
  - ➢ Move (7) after (8) and (9) to ensure both P1 and P3 can only print after both A and B are modified to the same value of X

```
        P1
[ 1] X = 1
[ 2] Y = 1
[ 3] while (T == 0);
[ 4] print A

        P2
[ 5] while (Y == 0);
[ 6] while (Z == 0);
[ 8] A = X
[ 9] B = X
[ 7] T = X

        P3
[11] X = 3
[10] Z = 2
[12] while (T == 0);
[13] print B
```

## Q1(d)
## Memory Consistency

- Would this modified program fail under relaxed consistency? If so, which particular models?
  - ➢ What instruction orderings are relaxed by each of the three RC models: TSO, PC, PSO?

- Answer
  - ➢ Does not fail under TSO and PC
  - ➢ Fails under PSO, as it allows $W \rightarrow W$ relaxation (can you give a sample execution scenario?)

Q2

## Relaxed Consistency

- Consider this shared memory program on 3 processors

**P1**

```
[ 1] X = 5
[ 2] Y = 2
[ 3] print Y
```

**P2**

```
[ 4] while (X == 0);
[ 5] Y = 6
[ 6] print Z
```

**P3**

```
[ 7] while (Y == 0);
[ 8] Z = 5
[ 9] print X
```

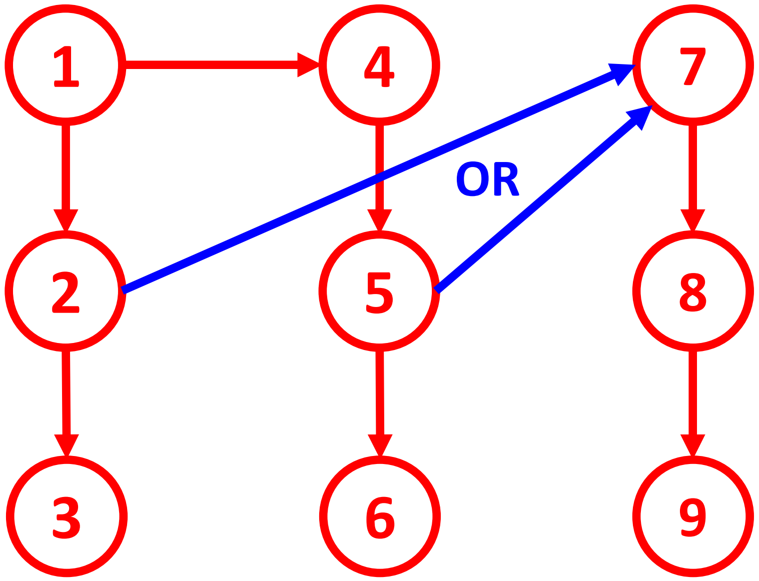
- This program was run on systems that utilize different *relaxed consistency models*, namely TSO, PC and PSO
  - All variables are initialized to **0**

```
        P1
[ 1] X = 5
[ 2] Y = 2
[ 3] print Y

        P2
[ 4] while (X == 0);
[ 5] Y = 6
[ 6] print Z

        P3
[ 7] while (Y == 0);
[ 8] Z = 5
[ 9] print X
```

Q2(a)

## Relaxed Consistency

- What can be inferred from the instruction dependency graph?



- Observe (7) can execute once execution of one of (3) or (5) becomes visible to P3

```
        P1
[ 1] X = 5
[ 2] Y = 2
[ 3] print Y

        P2
[ 4] while (X == 0);
[ 5] Y = 6
[ 6] print Z

        P3
[ 7] while (Y == 0);
[ 8] Z = 5
[ 9] print X
```

## Q2(a)
# Relaxed Consistency

- In one run, the following output was observed
  - $X = 5$, $Y = 2$, $Z = 0$

- Under which RC model(s) is this output valid, and why?
  - This is already valid under sequential consistency! (Can you give the execution scenario?)
  - Since SC is more stringent than all RC models, it is valid under all RC models (TSO, PC, PSO)

```
        P1
[ 1] X = 5
[ 2] Y = 2
[ 3] print Y

        P2
[ 4] while (X == 0);
[ 5] Y = 6
[ 6] print Z

        P3
[ 7] while (Y == 0);
[ 8] Z = 5
[ 9] print X
```

Q2(b)

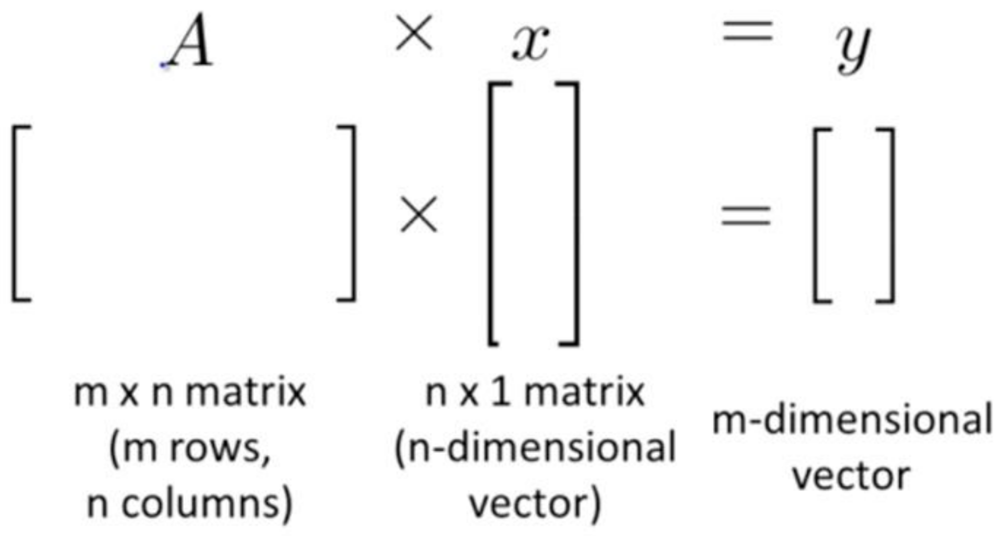
## Relaxed Consistency

- In one run, the following output was observed
  - $X = 0, Y = 2, Z = 5$
- Under which RC model(s) is this output valid, and why?
  - Only PC and PSO (invalid under TSO)
  - Notice $X = 0$ is printed by P3 $\Rightarrow$ $Y$ has been set by at least one of P1 or P2
  - $W \rightarrow W$ not relaxed: $Y$ set $\Rightarrow$ $X$ set by P1 $\Rightarrow$ $X = 5$ not seen by P3 (PC)
  - $W \rightarrow W$ relaxed: then $Y = 2$ set before $X = 5 \Rightarrow$ lets P3 print 0 (PSO)

Q3

## Parallel Programming Models

- Consider the problem of matrix-vector multiplication on a GPU with CUDA: $A \cdot b = c$
  - ➢ $\boldsymbol{A}$ **is an** $\boldsymbol{N \times N}$ **matrix and** $\boldsymbol{b, c}$ **are both vectors of size** $\boldsymbol{N}$
  - ➢ i.e. $m = n$ below



$A \times x = y$
m x n matrix (m rows, n columns)
n x 1 matrix (n-dimensional vector)
m-dimensional vector

## Q3(a)
## Parallel Programming Models

- How would you classify the memory model for CUDA: shared, distributed or distributed-shared? Explain with reference to the CUDA memory hierarchy.
  - Distributed-shared memory model
  - Different types of memory accessible at different scope
  - Distributed (thread-private): registers, local memory
  - Shared (block/kernel level): shared memory, global memory

Q3(b)

## Parallel Programming Models

- A kernel uses 64 registers and is launched with each block containing 512 threads. The kernel requires little local and shared memory and is run on a GPU device of Compute Capability 6.x with 65536 registers

- How many blocks and warps can reside on the GPU concurrently?
  - Blocks: $N_{blocks} = \frac{N_{registers}}{N_{registers/block}} = \frac{65536}{512 \times 64} = 2$
  - Warps: $N_{warps} = N_{blocks} \times N_{warps/block} = 2 \times \frac{512}{32} = 32$

Q3(c)

## Parallel Programming Models

- Discuss how you would implement matrix-vector multiplication in CUDA, detailing
  - Types of memory the kernel will use
  - Grid dimensions, block dimensions (and thus number of threads in a block)
  - Kernel code
  - Code that invokes the kernel

- In what ways can we decompose the problem into tasks to be assigned to CUDA threads?

Q3(c)

## Parallel Programming Models

- Considerations
  - Decomposition into <u>independent</u> tasks: each task as a multiplication/addition, row of A, or multiple rows from A
  - Dimensionality: for N tasks, 1D grid of $B = \frac{N}{T}$ blocks, each 1D block containing $T$ threads, depends on number of tasks
  - $T = 32i$ for some $i \in \mathbb{Z}^+$, for even division into warps
  - Kernel code - just a for loop, then write result to one cell of $c$
  - Bijective mapping from threads to a row/chunk of rows in A
  - Global memory: slow but cached in L1D$, coalesced txns
  - Shared memory: fast but poor performance if bank conflicts arise (especially for common vector $b$) $\rightarrow$ need strided access

Q3(d)

## Parallel Programming Models

- A CUDA implementation of the matrix-vector multiplication stores matrix $A$ and vectors $b, c$ all in shared memory. However, the size of the data exceeds the available shared memory

- How would you change your implementation to continue to use shared memory for large matrix sizes?
  - Only bring in sections of the required rows of the matrix $A$ into shared memory, i.e. stream matrix $A$ from global memory
  - Alternatively, launch kernel with more blocks of smaller size

Admin
## Tutorial Quiz

- Go to Luminus > Quiz
  - 1% of your grade
  - 8 minutes for 4 MCQ questions
  - **Don't be stressed - full credit for $\geq$ 50% correct (you should be able to do this after the tutorial, well, hopefully)**

- 10 minutes now to attempt it

# CS3210

## Parallel Computing

Thank you! Any questions?

## Tut 3

Mon (4pm)
Tues (2pm)