1. (a)

Data sharing	Message passing
Needs to have at least one portion of shared memory	Can be done with no shared memory at all
Requires cache coherency	Since data is shared via messages, each core sees only what other cores want it to see; no coherency required
Each memory access takes constant time (ignoring cache misses)	Memory access to data stored or controlled by another core takes time proportional to length of the path between the requesting and the fulfilling core. The time depends on the topology of the cores (e.g. fully connected, circle etc).

Both can (and do) use control structures such as locks, barriers etc, and both techniques can emulate each other.

(b) GPGPU devices have many simple cores that are optimised for simple operations such as graphics manipulations. They are designed to achieve a high throughput of floating point operations rather than a low latency on any one operation, and so they have lots of die area dedicated to ALU's, and comparatively less die area dedicated to control logic or caches than a CPU core would.

As a result of this massively parallel architecture, frameworks such as CUDA and OpenCL tend to let the programmer define kernels, which are then instansiated on many threads at the same time in the GPU. Each kernel can be thought of as a single iteration of a loop, with the iteration that is being executed is passed to the kernel at runtime.

Since accessing main memory from the GPU is very slow, data sharing is sometimes used, but not often used (memory access can easily become a bottleneck in GPGPU computation). Instead, message passing is commonly used as an alternative; the required data is copied into GPU memory (or a specialised GPU cache such as 'constant memory'), where the GPU can access it much more quickly.

When the GPU computation is done, the result is often copied back to main memory so it can be used by the rest of the program.

(c) Block scheduling is when a set region of iterations is given to each thread at compile time.

Cyclic scheduling is when iterations are taken from a queue and given to threads one by one at compile time.

Block-cyclic scheduling is the same as cyclic scheduling, but multiple iterations are assigned at a time, which can improve cache coherency, and reduce the overhead.

Static loop scheduling is preferred since the runtime overhead of the loop is reduced (sometimes to zero, but usually not since there is likely a barrier for the threads at the end). The alternative is dynamic loop scheduling, where threads are given the iterations they are required to compute at runtime.

(d) The load varies greatly per iteration; for most iterations, the work required is very small (just computing that j is not a square number and finishing), while in a minority of cases when j is a square number `procA` will be called which could take a (variably) long time. Thus, dynamically assigning work to threads is important so that the work can be distributed evenly.

Were threads to be assigned one iteration at a time, the overhead of this would be large (potentially larger than the work of the loop), mainly down to the fact that accessing the queue of iterations to be completed would be an atomic operation, which only one thread can be 'in' at a time. Even if a lock-free queue would be used, there would still be a relatively low upper limit on the number of iterations which could be assigned per second.

A better alternative would be to assign multiple threads at once so that the overhead is reduced. Balancing between the number of iterations assigned and how many long running iterations were given to the thread in that assignment might be tricky, since square numbers have an exponentially decreasing distribution. Compiler hints could help choose a sensible scheduling strategy so that good performance is observed at runtime.

N.b. You could refactor this code to have some threads finding which indexes need to call `procA` in a dynamic-block scheduled manner and add those indexes to a queue, and another set of threads

execuing `procA(a[x])` for the x's in the queue, who would be allocated one iteration at a time (since these iterations would take longer and the queue contention overhead would not be an issue).

2. (a) In order to coordinate multiple threads accessing data in the same address space, special hardware instructions must be used to do things like snoop on the activity of other cores and detect when a memory location has changed.

   Normal, high-level memory accesses in Java code will compile down to normal instructions like `LDR`, `STR` etc, so special Java functions are built into the language so that their less efficient, but synchronized counterparts can be used instead (such as load link/store conditional, or even transactional memory).

   These include classes such as `AtomicInteger`, `ReentrantLock`, `CyclicBarrier` and it's `AtomicReference` class that has the `compareAndSet` method.

   (b) Code when using normal instructions:

```
loop LDR R1, semaphore
     CMP R1, #0
     BNE loop
     ADD R1, R1, #1
     STR R1, semaphore
     BL  critical_section
     SUB R1, R1, 1
     STR R1, semaphore
```

   The reason why this doesn't work, is that multiple threads could read the same semaphore when it is not set, and enter the critical section at the same time.

   (Example would go here)

   Instead, the following code should be used:

```
loop LDR R1, semaphore ; Normal load first to avoid overhead
     CMP R1, #0
     BNE loop
     LDL R1, semaphore ; Load link the value
     BNE loop
     ADD R1, R1, #1
     STC R1, semaphore ; Store conditional
     BNQ loop          ; Revert if it failed
     BL  critical_section
     SUB R1, R1, 1     ; We can set the semaphore back with a normal instruction
     STR R1, semaphore
```

   (c) CISC style instructions (i.e. ones that do a lot of work for a single instruction) do not map well onto RISC machines with long instruction pipelines due to them taking multiple clock cycles to execute which can cause stalling.

   Furthermore, these instructions can require a CPU core to lock the snoopy bus for the duration of the instruction, which can degrade the performance of other cores in the processor.

   (d) Here is the semaphore using a CISC style TAS instruction:

```
loop LDR R1, semaphore ; Normal load first to avoid overhead
     CMP R1, #0
     BNE loop
     TAS semaphore     ; Test and set writes 1 to semaphore if
                       ; it is 0
```

```
     BNZ loop
     BL  critical_section
     MOV R1, #1
     STR R1, semaphore
```

The TAS instruction should not fail most of the time, but it is not ideal on RISC machines: The solution is to use load linked and store conditionals, which are more RISK-like instrucitons:

```
loop LDR R1, semaphore ; Normal load first to avoid overhead
     CMP R1, #0
     BNE loop
     LDL R1, semaphore ; Load link the value
     BNE loop
     ADD R1, R1, #1
     STC R1, semaphore ; Store conditional
     BNQ loop          ; Revert if it failed
     BL  critical_section
     SUB R1, R1, 1     ; We can set the semaphore back with a normal instruction
     STR R1, semaphore
```

Now the snoopy bus does not have to be locked since the load linked instruction requires observation of the snoopy bus rather than locking like test and set does. Store conditional does not interact with the snoopy bus at all, but merely checks to see if the load linked flag is still set in the CPU.

(e) Each thread decrements the memory address pointed to by `R2` in an atomic manner using load linked and store conditional instructions. They all then wait at what is effectively a barrier as they test the memory address pointed to by `R2` to see if it has the value 0 (this indicates that every thread has reached the barrier). When that is the case, then they can proceed to the line labelled done.

3. (a) Cache coherence refers to the need for each CPU cache to present the same view of system memory to each CPU core at the same time. This means that when one core updates a memory location, then each other core should be able to 'see' this update immediately.

   This is difficult to maintain as the CPU cores are connected by a bus, which has a limited bandwidth. As the number of cores increases (to n), the number of connections required to link them exhaustively increases following a square law (requiring n^2 links), which means the cost and complexity of the circuit increases. Alternatively, a single bus can be used, but this requires all memory reads and writes to be broadcast to the bus, meaning that it must handle n times the memory bandwidth.

   (b) Valid states are: MI (One core has written to a location) EI (One core has loaded the location, the other core hasn't) II (Perhaps DMA wrote to this memory location) SS (Both cores have an up-to-date copy of the location)

   (c) E is useful because it means that the core does not have to broadcast when it writes to the cache line, since it knows that other cores will not have to invalidate their cache line (since they do not hold it).

(d)

Time	State	Reason
1.	EII	Core one loaded the location
2.	SSI	Core two got the location from core one
3.	SSI	Nothing happened to memory
4.	SSI	Nothing happened to memory
5.	MII	Core one writes, which makes it modified, and invalidates core two
6.	SIS	Core three gets the value from core one, which is now shared
7.	SIS	Nothing happens to memory
8.	IMI	Core two writes, which invalidates the other cores
9.	IIM	Core three writes, which invalidates core two