Vor und Nachname (Druckbuchstaben): ________________________

Legi Nummer: ________________________

Unterschrift: ________________________

252-0024-00L
Parallele Programmierung
ETH/CS: FS 2014
Basisprüfung
Samstag, 09.08.14
120 Minuten

This exam contains 21 pages (including this cover page) and 8 problems. Check to see if any pages are missing. Enter all requested information on the top of this page, and put your Legi number on the top of every page, in case the pages become separated.

The only written aids you are allowed are 4 sides (2 A4 pages) of hand-written notes. You may *not* use additional notes, your books, or any calculator on this exam.

Read ahead, take five minutes to read through the questions.

The following rules apply:

- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit on problems where we ask you to show your working**. A correct answer, unsupported by calculations, explanation, or algebraic work will receive no credit; an incorrect answer supported by substantially correct calculations and explanations might still receive partial credit.
- If you need more space, use the back of the pages or the blank pages; clearly indicate when you have done this. **As a guideline, you should be able to answer the questions within the provided space.**
- Provide your answers either in English or German. Do not use a red pen!

Problem	Points	Score
1	17	
2	8	
3	10	
4	8	
5	10	
6	14	
7	22	
8	17	
Total:	106	

1. **Short Questions**

(a) (4 points) A single-threaded program you are developing is not fast enough. We assume that the program consists of a sequential part (that takes 60% of the execution time) and a parallel part (that takes the rest 40% of execution time). You have two options to make it faster:

1. Optimize the sequential part so that its time is cut down to half
2. parallelize the parallel part (assume that it scales perfectly)

Both approaches require the same effort. Under what conditions would you choose the second option? Justify your answer.

Solution: If t is the original execution time, then the first option cuts the time down to $t_1 = 0.4t + \frac{0.6t}{2} = 0.7t$. The second option results in an execution time of $t_2 = \frac{0.4t}{C} + 0.6t$, where C is the number of cores. if $t_2 < t_1$, then we can select the second option. That is, $C > 4$ cores. So we need to have more cores than 4 to take the second option. (Grading: 1pt for just giving Amdahl's law, 1pt for each formula, 1pt for the conclusion of which to prefer based on number of cores.)

(b) (2 points) Consider you have to build an application that handles all incoming requests in a web server. For processing requests you have a choice between threads and tasks. Which one would you use and why?

Solution: (Grading: 2pt for correct answer AND reasonable argumentation.)

Tasks: fast context switch, cheap to create, can create many of them to enable fine-grained parallelism, matches with short lifetime of web requests.

Threads: more expensive to create, reuse a pool of threads (sized to match no. of cores), apply work stealing to achieve load balancing.

Non-answers: "can take advantage of parallel processing" (too vague), "tasks threads share address space" (applies to both tasks and threads).

(c) (2 points) Name two approaches to reduce lock contention and briefly explain the effect each of them have.

Solution: (Grading: 1pt for each correct argument)

1. Reduce duration that locks are held, e.g. using smaller synchronized blocks;
2. Reduce frequency of lock requests, e.g. via lock splitting or striping;
3. Replace exclusive locks with coordination mechanisms that permit greater concurrency, e.g. Reader/Writer locks, non-blocking data structures etc.
4. Split up a lock into smaller locks (fine-grained locking)
5. Avoiding locks entirely: lock-free data-structures, replication, immutability etc.

Also accepted: conditions instead of busy-wait, transactional memory, fairness model, faster lock implementation (if sufficiently justified).

(d) (3 points) Explain in words what a barrier is and describe its interface. eschreiben Sie in Ihren eigenen Worten was eine *Barriere (eng.: barrier)* ist und wann Sie dieses Konstrukt benutzen würden. (Verwenden Sie ggf. ein Beispiel).

Solution: Barrier: rendezvous for arbitrary number of threads. Alternative definition: A barrier is a way of forcing asynchronous threads to act almost as if they were synchronous.

Interface: `init(N), await(), reset()`

If they do not mention reset, it's OK.

1p for correct definition, 2p for mentioning the API or pseudo-code of how to use.

(e) (3 points) In Java there are two types of thread-safe collections: synchronized (Collections.synchronizedMap) and concurrent (e.g. ConcurrentHashMap). Explain the difference and give a benefit of concurrent over synchronized.

Solution: Synchronized (i.e. lock based wrappers of single-threaded implementations) Concurrent (i.e. designed for concurrency)

"A concurrent collection allows multiple threads to be reading/modifying it in parallel, whereas a synchronized collection does not. On the other hand, it is hard to make compound operations atomic (e.g., of the flavor read-modify- write), since you cannot use client-side locking"

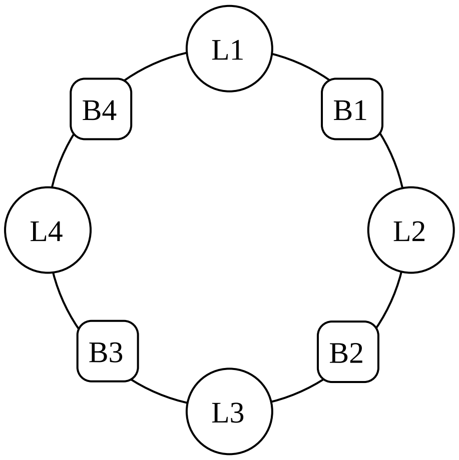
2p for explaining the difference; 1p for stating the advantage of concurrent over synchronized.

(f) (3 points) Briefly compare and contrast shared vs. isolated mutability.

Solution: In the former, state is shared and modified by multiple threads and needs to be protected by synchronization primitives. With the latter, state is still mutable but it is now private to each thread/task and they cooperate with message passing (see lecture 17, slide 7 onwards).

2. (8 points) **Controlling Lights**

   Assume you have 4 Lights (`L`) and 4 Buttons (`B`) arranged in a circle as in the figure below. The lights are represented by the `Light` class, whose fields are protected by the intrinsic lock (see for example `setColor()`).



   Create a function `tryPressButton()` that presses the button only when **both** neighbor lights are GREEN. Note that there are hints in the comments of the code below. Other threads can operate on the `Light` objects (e.g., by calling `setColor()`). Explain your synchronization scheme using a few sentences.

```
enum Color { // Different colors
    GREEN,
    RED,
    ...
}

public class Light {
    // for simplicity, fields are public
    public int id;      // light number (1, 2, 3, 4);
    public Color color; // current color;

    // class is protected by the intrinsic lock
    public void synchronized setColor(Color c) {this.color  = c;}

    ...
}

// You can use the following functions:
// get the i light. Assume this function is thread-safe
public static Light getLight(int i)  { ... }
// get the i button. Assume this function is thread-safe
public static Button getButton(int i) { ... }

// You need to complete this function:
public void tryPressButton(i) {

    // Get the ith Button
    // You can use b.press() to press it
    Button b = getButton(i);

    // get the ith Light
    Light li = getLight(i);
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
}
```

Explanation of synchronization scheme:

Solution:

```java
public static void tryPressButton(int i) {
    Button b = getButton(i);

    Light l1 = getLight(i);
    Light l2 = getLight(i+1 % 4);
    Light fst, snd;

    if (l1.id > l2.id) {
        fst = l1;
        snd = l2;
    } else {
        fst = l2;
        snd = l1;
    }

    synchronized (fst) {
        synchronized (snd) {
            if (fst.color == Color.GREEN &&
                snd.color == Color.GREEN)
                b.press();
        }
    }
}
```

3. **Iterative algorithm**

   (a) (7 points) Assume you have an initial array A_0 of size N, and an algorithm that takes R repetitions, and in each produces a new version of the array based on the old version. Each element in the new array depends **only** in the old array and the location of the element (`fn` function in the pseudocode example below).

   Pseudocode:

```
for (r=0; r<R; r++) {    // r: repetitions
  for (i=0; i<N; i++) { // i: array element
    $A_{r+1}$[i] = fn($A_r$,i) // compute ith element
  }
}
```

   This algorithm can be parallelized as follows:

```
for (r=0; r<R; r++) {
  create T threads
  give each thread a part of the output array to compute
  wait untill all threads are finished
}
```

   Where each thread works as follows:

```
// perform the computation for the given part of the array
for (/* thread's part of the array */) {
  $A_{r+1}$[i] = fn($A_r$,i)
}
```

   Spawning the threads at each step, however, has overhead. Describe the modifications needed to provide a **correct** version of the above code, where the threads are spawned only in the beginning. You can use the skeleton below.

   Algorithm:

```
allocate all arrays
.....................................................
.....................................................
create T threads
.....................................................
.....................................................
give each thread a part of the output array to compute
.....................................................
.....................................................
wait until all threads finish
.....................................................
.....................................................
```

Each thread:

....
....
....
....
....
....

Solution: (Grading: full points if barrier mentioned, minus points if something else wrong, partial points for stating reuse of threads (2pt), otherwise zero.)

Algorithm:

```
allocate all arrays
create barrier initialized at T
create T threads
give each thread a part of the array
wait until all threads finish
```

Thread:

```
for (r=0; r<R; r++) {          // r: repetitions
  for (/* part of the array */ ) { // i: array element
    $A_{r+1}$[i] = fn($A_r$,i)  // compute ith element
  }
  barrier();
}
```

(b) (3 points) Assume that f(A_r, x) takes time k for $x \in 1, 2, \ldots, n$ etc. And $f(A_r, x = 0)$ takes time $20 \times k$. Would that cause a problem? Justify your answer and briefly describe a solution to the problem if one exists.

Solution: Problem: load balancing (1pt). Solution: dynamic scheduling (2pt).

We defined fixed costs so static scheduling is also fine – i.e. assign proportionally less work to the thread processing the first element and adjust among the remaining threads. Not accepted: calculate f(A_r, $x = 0$) sequentially and only then compute the rest in parallel.

4. **Data and pipeline parallelism** Assume you have functions `f()`, `g()`, `h()` that take a single `Integer` argument, and return an `Integer`. Also assume that they **only** access their argument and no other data when executed.

   For example:

```
public Integer f(Integer x) {
        // only access x, and no other data
}
```

   You want to compute `f(g(h(x)))` for each element `x` on a large array.

   (a) (2 points) Describe in a few sentences how can you build a parallel version of this computation using the data parallel model.

   Solution: This is a map. An acceptable answer is also to split the array and distribute chunks to threads.

   (b) (2 points) Describe in a few sentences how can you build a parallel version of this computation using a pipeline.

   Solution: One thread per function.

   (c) (2 points) What is the maximum speedup you can get when using the pipeline version and why? (only consider the stages where the pipeline is full)

   Solution: We have three stages. At best, if the they take the same time we can get a speedup of 3.

   (d) (2 points) Can you combine the pipeline and data parallel versions? Give an example.

   Solution: Yes. each pipeline stage in parallel, or multiple pipelines.

5. (10 points) **Matching Parentheses**

In this task we are concerned with the problem of matching parentheses. We would like to count the number of *unmatched* open and closing parentheses in a string. For convenience, the string is represented by an array where 0 is an open parenthesis and 1 is a closing parenthesis. Some examples and the expected output are shown below.

String	**Array**	**Result**
(	[0]	open = 1, closed = 0
)(	[1,0]	open = 1, closed = 1
()(	[0,1,0]	open = 1, closed = 0
()()	[0,1,0,1]	open = 0, closed = 0
(((())	[0,0,0,0,1,1]	open = 2, closed = 0
))()()(	[1,1,0,1,0,1,0]	open = 1, closed = 2

Sketch how you would implement a method `matchParenParallel(int[] array)` that takes the entire array and computes its result *in parallel*. You can assume there is already a method for matching parentheses sequentially (as shown below) which you are free to use as a subroutine if you wish.

```
public static Pair<Integer, Integer> matchParentheses(
    int[] array, int start, int end) {
  int open = 0, closed = 0;
  for (int i = start ; i < end; i++) {
    // (Details of how this is implemented don't matter.)
  }
  return Pair.of(open, closed);
}
```

You can provide pseudocode, text, and/or drawings in your explanation. If you use any frameworks (e.g. raw threads, Java standard library, MapReduce) make sure it is clear from your explanation how these would be used.

Solution: Some possible answers include:

- **Fork-join** Basically you will keep dividing the array in half, processing it in separate tasks until it falls below a certain minimum size. When the array is below the minimum size, you call `matchParentheses` on it to get the answer. Otherwise, you combine the two results of the two children, as follows. Let the result be a 3-tuple: `(closed, openL, openR)` where `closed` means a matched pair "(. . .)", `openL` is "(" and `openR` is ")" then:

```
    // Balanced pairs -> can be eliminated (which is why we subtract).
    newClosed = min(tupleL.openL, tupleR.openR)
    // Same number as before, plus the new matched pairs.
    closed = newClosed + tupleL.closed + tupleR.closed
    // Accumulate unmatched parentheses, but deduct the count above.
    openL = tupleL.openL + tupleR.openL - newClosed
```

```
        openR = tupleL.openR + tupleR.openR - newClosed
        return (closed, openL, openR)
```

- **(Static) work partitioning** Divied the array into N parts, where N is the number of processors or something larger if the array is too small (so chunks stay above a minimum size). Then you would call `matchParentheses` on each of these in parallel, and then combine the results, left to right.
- **Map-Reduce** It is more clumsy, since mapping tasks emit using key-value pairs. Probably you would have a single key for the output, and so a single reduce task would handle the list of results associated with that key. This list would have to include an identifier to associate each result with its position in the original array. (Some people tried this, but very few of those answers were correct or even detailed enough.)

Grading:

- -6pt: no merge
- -4pt: wrong merge (no adjust, missing `join/barrier`)
- -2pt: adjust is not fully correct
- -4pt: inefficient but correct (e.g. pruning)
- -2pt: no cutoff / sequential call
- -3pt: not explaining how framework is used
- -1pt: one thread per element

6. **Data parallelism** Java 8 introduced a new framework to transform a collection of elements. As a reminder of some of what the API offers:

```
interface Stream<T> {
  /* Returns the count of elements in this stream. */
  long count();

  /* Returns only elements that match the given predicate. */
  Stream<T> filter(Predicate<? super T> predicate);

  /* Transforms a stream by applying the given function to every
     element. */
  <R> Stream<R> map(Function<? super T,? extends R> mapper);
  ...
}
```

(a) (2 points) Explain one pro and one con of the design of this API. (As a comparison point you can consider normal for-loops and threads.)

Solution: (Grading: 1pt each for mentioning an advantage and disadvantage.)
Pros: declarative, no storage, functional, easy to parallelize.

Cons: less flexibility than for-loop (e.g. using neighbouring elements), operators must be stateless.

There is also more in-depth discussion here: `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

(b) (4 points) A bank has faulty ATM machines which sometimes writes a transaction multiple times to the log and so they wrote a routine to drop an element when it is equal to its neighbour. For this they used the new Streams API and their solution is fine when run sequentially, but doesn't work when parallelized. Explain the problem in 2-3 sentences.

```
List<String> accounts = Arrays.asList(new String[]{ "C3",
   "B2", "B2", "A1", "C3", "D4", "D4", "D4" });
// Expected output: ["C3", "B2", "A1", "C3", "D4"]

String last = null;
List<String> result =
  accounts.parallelStream()
          .filter(x -> {
              if (x.equals(last)) {
                return false; // drop
              } else {
                last = x;
                return true; // keep
              }
          })
          .collect(); // create a collection from the string
```

Solution: Problem: code depends on the order of execution (1p), however when stream execution is parallelized there is a race condition (1p) to access the shared state which leads to non-deterministic results (1p). More generally, the streams model has the limitation that operators should be stateless (1p).

Note 1: just wrapping all uses of the `last` variable in `synchronized` blocks does *not* fix the problem. The variable itself needs to be thread-local and you still need to take care of merging boundaries.

Note 2: `filter` does not delete or modify the underlying collection so that is not a valid ground for explaining why data races can occur.

(Also: some mentioned you *cannot* parallelize due to the dependency between loop iterations, yet they answered part (c) correctly ;-)

(c) (8 points) Lets simplify the previous problem; you just need to detect that there are duplicates, without removing them. Using the fork-join framework write a function which does the following: given a list/array, return true if any pairs of elements are equal, otherwise return false. (Pseudo-code is fine and you can use the `spawn` and `sync` keywords as in the lecture.)

Solution: Spawn until you reach single elements and compare if these are equal. Can exit early if this is true. After this, sync on the two sub-halves and at every merge step check if the boundary elements are equal.

Grading: we checked sanity of the algorithm, cutoff, split/spawn, merge/join.

- -1pt: not using the most efficient solution (checks against all elements)
- -1pt: for not merging the results
- -1pt: no base case
- -1pt: does not check the boundary elements
- -1pt: bad recursion step
- 1pt for just explaining the fork-join paradigm, but not applying it to the problem at hand

7. **Implementing Locks**

(a) (4 points) When waiting in a synchronization primitive, there is the possibility to spin or suspend execution. Explain these two options and the trade-off on performance.

Solution: (Grading: 2p for explaining both options; another 2p each for a suitable justification of when each is appropriate. Common reason for not receiving full points was not mentioning queue/notification and latency implications.)

Busy-wait: continuously check a value until it changes. Wastes CPU execution cycles.

Suspend execution while you wait. Relies on a notification mechanism, so typically support from the OS scheduler is required. Does not waste CPU time but they have higher wakeup latency.

(For more, see the beginning of lecture 12, which explains busy-wait and the implications for uncontended vs. contended access.)

(b) (8 points) You are building a new machine which only provides hardware support for a single thread synchronization primitive: *compare-and-swap* (CAS). As a reminder, this is an atomic operation and its signature and semantics are described below.

[NOTE: Java use different naming ... add a note that compare-and-swap and compare-and-set are one and the same]

```
class AtomicBoolean {
  /**
   * Atomically sets the value to the given updated value
   * only if both the current value and the expected values
   * are equal.
   *
   * @param expect the expected value
   * @param update the new value
   * @return True if successful; False return indicates that
   * the actual value was not equal to the expected value.
   */
  boolean compareAndSet(boolean expect, boolean update);

  /* Returns the current value. */
  public final boolean get();

  /* Unconditionally sets to the given value. */
  public final void set(boolean newValue);

  ...
}
```

Your task is to implement the code for a simple spinlock using only this atomic operation and nothing else. The lock you implement does *not* need to be reentrant and you can assume the client code uses the lock correctly (e.g. no double release or release by a thread not holding the lock).

```
class SpinLock {
  // Define any new members you need here.
  private final AtomicBoolean locked  = new AtomicBoolean();
  .......................................................
  .......................................................

  public void acquire() {
    .......................................................
    .......................................................
    .......................................................
    .......................................................
  }

  public void release() {
    .......................................................
    .......................................................
    .......................................................
    .......................................................
  }
}
```

Solution:

```
class SpinLock {
  private final AtomicBoolean locked  = new
     AtomicBoolean();

  public void acquire() {
    while (!locked.compareAndSet(/*expect*/ false,
       /*update*/ true)) {
      /* spin */
    }
  }

  public void release() {
    locked.set(false);
  }
}
```

(c) (10 points) After benchmarking your implementation, you realize that spinning is inefficient when you have to wait for a long time. The hardware designers tell you they can add two more primitives to make the lock more efficient:

- `atomicSleep` is an atomic operation which corresponds to the following pseudo-code:

```
// Note: SuspendLock is discussed below
void atomicSleep(SuspendLock l, AtomicBoolean val,
   boolean x) {
  atomic { // atomically:
    val.set(x); // set value to x
    suspend(l); // stop (suspend) the thread.
                // the thread is now sleeping on lock l.
  }
}
```

- `wakeUp` is non-atomic with the following pseudo-code:

```
void wakeUp(SuspendLock l) {
  wakeup_thread(l); // wake up a thread sleeping on lock l
}
```

Your second task is to reimplement the lock more efficiently using these operations. You may recognize this matches the basic design of a mutex. (The same assumptions hold as in the previous part.)

```
class SuspendLock {
  // Define any new members you need here.
  .......................................................
  .......................................................
  .......................................................
  .......................................................
  .......................................................
  .......................................................

  public void acquire() {
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
  }

  public void release() {
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
    .......................................................
  }
}
```

Solution:

```
// Note: value does not have to be atomic I think. It's
   just a value that
// signifies if the lock is taken. It is prortected by the
   guard spinlock.
class SuspendLock {
  private final AtomicBoolean guard  = new AtomicBoolean();
  private final AtomicBoolean value  = new AtomicBoolean();

  public void acquire() {
    while (true) {
      while (!guard.compareAndSet(false, true)) { /* spin
         */ }
      if (value) {
        atomicSleep(this, guard, false);
      } else {
        lock.value = true;
        lock.guard = false;
        return;
      }
    }
  }

  public void release() {
    while (!guard.compareAndSet(false, true)) { /* spin */
       }
    lock.value = false;
    wakeUp(this);
    lock.guard = false;
  }
}
```

8. **OpenCL**

(a) (2 points) Recall Flynn's taxonomy; under what class would you place (i) sequential CPUs and (ii) GPUs?

Solution: Sequential CPU: SISD – Single Instruction, Single Data (1p) Vector GPUs are SIMD – Single Instruction, Multiple Data (1p)

(b) (3 points) Compare CPUs and GPUs in terms of throughput and latency. Justify these performance characteristics in terms of architectural differences.

Solution: CPU:

- Rich instruction sets with support for loops, conditions.
- Fewer execution units but highly tuned for single-threaded performance (e.g. out-of-order and speculative execution).
- Optimised for low latency access to cached data sets.

GPU:

- Data-flow architecture, inherently SIMD – a single operation can operate on many elements in parallel.
- Architecture tolerant of memory latency.
- More transistors dedicated to computation.
- Throughput-oriented (aim for peak performance).

For a very thorough discussion of this topic see this document: Palacios, Jonathan, and Josh Triska. "A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both." (2011). `http://cours.do.am/ParadigmeAgent/final.pdf`
2p for CPU and 2p for GPU. 1p each for mentioning any these or other well-justified answers.

(c) (5 points) GPUs have several different types of memory as shown in the figure below. Explain what each of these are by discussing access rights, where these are physically located and the impact on access latency.

Note: In your answer you should mention: private, local, global, constant and host memory.

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

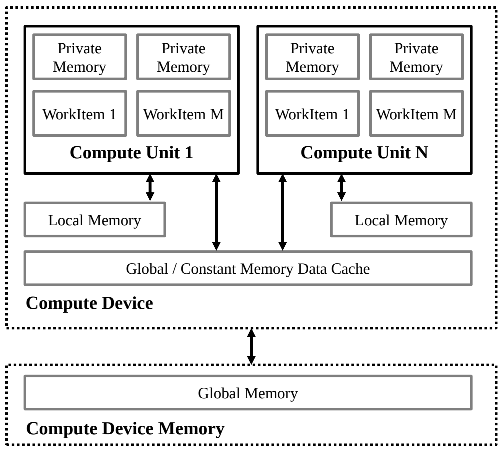


Figure 1: GPU Memory Hierarchy

Solution:

- Private Memory: Per work-item
- Local Memory: Shared within work-groups
- Global / Constant Memory: Not synchronized, latter is read-only
- Host Memory: On the CPU

1p for explaining each of these. (For more, see also: lecture 21, slides 5 and 6)

(d) (3 points) Assume you are writing an algorithm that requires a tree reduction. The input array is too large to fit within a single work-group and the final result needs to written to global memory. How would you make sure the tree reduction is thread safe. Hint: you can assume the reduction operation is associative and commutative.

Solution: Each work-group performs a partial reduction a barrier is used to ensure all work items have finished the first item in the work group writes the partial result to memory. A new kernel (launched from the host, once all work-groups have finished) collects the partial results and performs the final reduction. (This can then be performed in a loop if necessary).

(e) (4 points) Below is a simple OpenCL kernel that uses a 2D grid of work-groups. It reads from *global* memory and fills a chunk of local memory. Finally, the work-item computes the final result by adding the values read from two *local* memory addresses.

Give an explanation for why the memory fence is needed in this code.

```
__kernel void fill_tiles(__global float* a,
                         __global float* b,
                         __global float* c)
{
  // find our coordinates in the grid
  int row = get_global_id(1);
  int col = get_global_id(0);

  // allocate local memory shared among the workgroup
  __local float aTile[TILE_DIM_Y][TILE_DIM_X];
  __local float bTile[TILE_DIM_Y][TILE_DIM_X];

  // define the coordinates of this workitem thread
  // in the 2D tile
  int y = get_local_id(1);
  int x = get_local_id(0);

  aTile[y][x] = a[row*N + col];
  bTile[y][x] = b[row*N + col];
  barrier(CLK_LOCAL_MEM_FENCE);

  // Note the change in tile location in bTile!
  c[row*N + col] = aTile[x][y] + bTile[y][x];
}
```

Solution: Work-items need to be synchronized before writing to global memory as values in local memory might not be visible to all work-items. For example, the part of local memory that is read by one work-item might not have been filled yet.

(4 points: barrier, local memory fence, global memory fence and then an explanation.)