CSE113 – Spring 2021
Final
Assigned: June 9 (by 12:01 AM)
Due: June 9 (by 11:59 PM)

Before you begin: This is an asynchronous exam, and you should treat it like such. Please do not discuss the exam with any other student until after the due date. This exam is open book and open notes. These resources should be sufficient to complete the test.

The test is "open internet", to a degree. You are allowed to search online for general concepts. You are not allowed to search for the exact questions. You are especially NOT allowed to ask test questions on forums like StackOverflow, Reddit, or Quora. There is a zero tolerance policy on cheating. The department does monitor these sites (and others). Students have been reported for academic misconduct.

If you have clarifying questions: Please write a question to the teach mailing list, or an instructor only question on Piazza. We will keep a thread of clarifying questions as a Canvas discussion. Please check there before asking your question. While there are 12 hours to complete the test, we allocated time for our test is 4 PM to 7 PM. We will be most active answering questions during that time. There is no guarantee that any questions will be answered after 7 PM.

There are 6 questions, each worth equal points. The exam is cumulative: any material covered in class or homework is possible to appear on the exam. The test is designed to take 2 hours, assuming that you have spent enough time studying and do not double check your answers (like what would happen in a traditional synchronous exam). However, within the available hours, you are free to spend as much time as you'd like on the test.

I will provide the MS word doc, as well as well as the PDF. Your submission should be in PDF form, but we are flexible on the format. You can print it out and fill in the test with pencil/paper and scan/photo your solutions. You can work in the MS word document. As long as your solutions are clear to us. Please show your work.

Good luck!

Question 1: *Matrix multiplication loops*

The program in this question is matrix multiplication. It uses 2D arrays (i.e. matrices), which are indexed using two values. The first value specifies the row, and second value specifies the column. In C/C++, these arrays are stored in row-major style. That is, rows are contiguous in memory. To be more explicit, consider a matrix `A` with dimensions (`d0,d1`). Accessing `A[i][j]` is equivalent to accessing a static 1D array with index `[i*d1 + j].` You should note that accessing elements in the same row that are next to each other will provide good cache locality, while accessing elements in different rows (even if the rows have contiguous indexes) will have poor cache locality.

Assume the input is two arrays to multiply: `A` and `B`, with the dimensions of `A` being (`d0,d1`) and the dimensions of `B` being (`d1,d2`). The result is stored in `output` with dimension (`d0,d2`). The code works by iterating over each row of `A` (the outer `i` loop), and each column of `B` (the middle `j` loop), and computing a dot product (the innermost `k` loop). The code is given as follows:

```
for (int i = 0; i < d0; i++) {
  for (int j = 0; j < d2; j++) {
    for (int k = 0; k < d1; k++) {
      output[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

You can assume d0, d2, and d1 are all positive and greater than 0. You can also assume they are sufficiently large (e.g. larger than a cache line).

a) For each of the 3 nested loops, specify which ones (if any) are safe to do in parallel (i.e. which loops are DO-ALL loops). Justify your answer.

b) Assume the input matrix `A` is given transposed, then the only difference from the original kernel would be that the 4th line would change to the following:

```
output[i][j] += A[k][i] * B[k][j];
```

Describe in a few sentences the performance effects this will have. Note that the computation is valid regardless of the nesting of the for loops. Describe a loop nesting order that will provide better performance and justify your answer.

c) Going back to the original code of part (a). Assume you are constrained such that you can only parallelize the inner-most loop (the loop over `k`). However, you know that there will be exactly 4 threads; and you are allowed to use global variables. You are also allowed to perform the dot product accumulation in any order (your solution is allowed floating point errors that might arise due to the order of operations). You can assume that `d1` is evenly divisible by 4.

Write a solution based on the code snippet above that launches and joins C++ threads to perform this loop in parallel. It should NOT use atomic variables, it should NOT have any data-conflicts. It should compute the same result as the reference code loops, allowing for small differences due to floating point order-of-operation differences.

Question 2: *Semaphores*

A special type of mutex is called a *Semaphore.* It is a mutex that allows N threads into a critical section. A mutex can be seen as a special case of a semaphore, where N == 1. A common analogy is a bus: a certain number of people are allowed on the bus (up to N), and as some people get off the bus, more are allowed on (again, up to N). Note that threads are allowed into (and out of) the critical section asynchronously. That is, all of them do not have to arrive, or leave, at the same time.

Your job is to implement a semaphore. Your class should contain atomic variables or mutexes, but you are not allowed to use conditional variables or the C++ built-in semaphores. As a hint, look at the reader-writer mutex of homework 2.

a) Implement the 3 API calls: the constructor, `enter`, and `exit`: each are documented with comments below. You can add private variables or functions if needed (according to the above constraints). The signature of the public API calls cannot be modified.

```
class Semaphore   {
 private:
    // give me private variables
 public:
  Semaphore (int num_allowed_threads) {
    // Implement me
  }

  void enter(int tid) {
    // Up to N threads (specified in the constructor)
    // are allowed to enter.
  }

  void exit(int tid) {
    // threads that exit should make room for more threads to enter.
  }

};
```

b) implement a new member function:

```
void resize(int new_num_allowed_threads)
```

this function can be called concurrently with `enter` and `exit` and will change the number of allowed threads in the semaphore. The function should block until it can successfully complete.

c) implement two safeguards in the solution of part (a). A thread that has successfully called `enter` is not allowed to call `enter` again before calling `exit`. Similarly, a thread should not call `exit` unless it is currently inside the semaphore's critical section (i.e. successfully called `enter` earlier). You can assume the number of threads is capped at a compile-time constant `NUM_THREADS`. You should modify the API to return an integer: 0 on success, -1 on failure.

There are many ways to implement this check. The ideal solution should provide only O(1) time overhead in the enter and exit functions. You are allowed to use the C++ standard library, however, an elegant solution exists just using a static array.

Question 3: *Concurrent Data-structures*

In class we discussed a concurrent linked-list based set. Here we will discuss a simpler set. This set is grow-only, i.e. it supports only adding elements, but not removing them. It can contain only positive integers. It is based on a static array. You can assume the user will not add any more elements than the underlying array can hold.

The set is initialized with the underlying static array (`elements`) holding the sentinel value -1 in all locations. That is, -1 means that the location is unused. It also keeps track of an index where a new element can be added (`end_index`). The `contains` method iterates through the array to check if a value is in the set. The `add` method first checks to see if the value is contained in the set. If not, the new value is added to the list and the `end_index` is incremented. The sequential specification is given below:

```
class AddOnlySet {
private:
  int end_index;
  int* elements;

public:
  AddOnlySet(int s) {
    end_index = 0;
    elements = new int[s];
    for (int i = 0; i < s; i++) {
      elements[i] = -1;
    }
  }

  bool contains(int check) {
    for (int i = 0; i < end_index; i++) {
      if (elements[i] == check) {
        return true;
      }
    }
    return false;
  }

  void add(int to_add) {
    if (contains(to_add)) {
      return;
    }
    else {
      elements[end_index] = to_add;
      end_index++;
    }
  }
};
```

a) Recall that thread safety means that a program is free from data-conflicts and concurrent behavior is well-specified. As written, is this data structure thread-safe? If the private variables are changed to atomic variables, is the data-structure thread safe? If not, give an example where the concurrent calls to the public methods (`add` and `contain`) could cause a problematic result according to the specification of the set.

b) Add one or more mutex private variables, and call `lock`/`unlock` in the methods to make the class thread safe. You should not make the other private members atomic. You can add private methods if you'd like, but the API to the public methods must remain unchanged. *Note that a data-conflict, i.e. concurrent accesses (where at least 1 access is a write) to a non-atomic variable is NOT thread safe, even if it isn't obvious how the compilation and execution can lead to an incorrect result.*

Justify in a few sentences why your implementation is thread safe. Use your example in part (a) and explain why it can no longer occur.

c) Remove the mutex(es) from part (b) and make the class both thread-safe and lock-free. You are limited in that the only RMW you can use is the C++ `compare_exchange_strong` (i.e. what we used in the spin-locks).

As some hints: you will need to change the existing private variables to atomic. You do not need any additional private variables. Recall that if the C++ `compare_exchange_strong` fails (i.e. the comparison is false), it returns false, and the value in memory is returned through a pass-by-reference argument. Your implementation should not allow duplicates in the `elements` array.

Justify in a few sentences why your implementation is correct, and explain why your example in part (a) is no longer allowed.

Question 4: *Memory consistency models*

In class, we discussed a way of specifying memory models in terms of a grid that specifies which type of memory accesses can be re-ordered with other types of memory accesses. Recall that the special `FENCE` instruction can be used to restore orderings. Consider a new architecture with the following ordering specification:

		memory access 0: Load	memory access 0: Store
memory access 1	Load	different address	different address
	Store	NO	NO

If memory access 0 appears before memory access 1 in program order, can it bypass program order?

for a,b,c: given a litmus test, determine if the specified outcome is possible. Justify your answers with an execution trace (i.e. Lego stack). Additionally, if the outcome is allowed, describe if you can place a FENCE in the program to disallow the behavior

a)

```
Global variables:
int x[1] = {0};
int y[1] = {0};

Thread 0          Thread 1
t0 = load(x)      t1 = load(y)
store(y,1)        store(x,1)

At the end of the execution can:
t0 == 1 and t1 == 1?
```

b)

```
Global variables:
int x[1] = {0};
int y[1] = {0};

Thread 0          Thread 1
store(x,1)        store(y,1)
t0 = load(y)      t1 = load(x)

At the end of the execution can:
t0 == 1 and t1 == 1?
```

---

c)

```
Global variables:
int x[1] = {0};
int y[1] = {0};

Thread 0          Thread 1
t0 = load(y)      store(x,1)
t1 = load(x)      store(y,1)

At the end of the execution can:
t0 == 1 and t1 == 0?
```

---

d) You are developing a compiler from C++ to this new architecture and are implementing the translation from the default C++ atomics (sequentially consistent) to this new architecture. Describe how you would translate the following 2 C++ instructions to pseudo-ISA instructions. The pseudo-ISA instructions you can use are `load()`, `store(...)`, and `FENCE`. Your translation can include more than one instruction (i.e. a memory access proceeded or preceded by a FENCE) Please justify your answers.

- for an `atomic_int x`: how would you translate `x.load()`
- for an `atomic_int x`: how would you translate `x.store(...)`

Question 5: *Fairness*

In this question we will consider the following program. You should assume sequential consistency.

```
Global variables:
atomic_int x(0);

Thread 0          Thread 1
x.store(1);       while (x.load() == 1);
x.store(0);
```

a) Draw a labeled transition system (LTS) graph for this program.

b) Is there a potential for non-termination due to starvation when executed under an unfair scheduler? if so, describe how this might happen.

c) Is the program guaranteed to terminate under the non-preemptive scheduler? i.e. once a thread has been scheduled to a core, it will continue fairly executing. Please justify your answer.

d) Is the program guaranteed to terminate under the energy-saving scheduler (also known as the HSA scheduler)? As a reminder, this scheduler might unfairly preempt threads, but guarantees fair execution to the thread with the lowest id. Justify your answer.

e) Consider swapping the thread programs, i.e. thread 0 now spins while x is equal to 1; thread 1 will write 1 followed by a write of 0 to x. Do your answers to c and d hold? Explain your reasoning.

Question 6: *GPUs*

a) Consider the blur function you worked on in Homework 4 (note that this blur is not repeated, it is only done once). The C++ code is given below. Your job is to write a CUDA program to do this function on the GPU, and in parallel. Compute your GPU blur using 1 block with 16 warps.

```
void blur (double *input, double *output, int size) {
  for (int i = 1; i < size-1; i++) {
    output[i] = (input[i-1] + input[i] + input[i+1]) / input[i];
  }
}

int main() {
  double *input = new double[SIZE];
  double *output = new double[SIZE];

  // input initialization (not shown)

  blur(input, output, SIZE);

  // check results in output (not shown)

  return 0;
}
```

b) Unrelated to part (a), but as we discussed in class, the indexing pattern (e.g. round-robin or chunking) that threads use in a GPU kernel can have a large effect on the performance of the GPU kernel. That is, if threads that share a load/store unit access different memory locations, the load/store unit might have to make more (or less) requests to the GPU memory.

Assume a GPU kernel running with a single warp (32 threads); assume 4 contiguous threads share a load/store unit. For example threads 0,1,2,3 share a load/store unit, threads 4,5,6,7 share a load/store unit, etc.

If all threads access some input array ***ONCE***, accessing a 32 bit integer type, using the following indexes, please describe how many memory requests the load/store units across the entire warp must make to the GPU memory. You should assume that the input array is large enough such that the indexes never go out of bounds. You can assume that the load/store unit fetches 16 bytes at a time.

- if threads index memory using: `threadIdx.x` (the GPU thread id)
- if threads index memory using: `threadIdx.x + 4`
- if threads index memory using: `threadIdx.x / 4` (integer division)
- if threads index memory using: `threadIdx.x * 2`
- if threads index memory using: `threadIdx.x * 4`