15-418/618 Spring 2020
Exam 2

Released:	Wed., April 15	12:00 Noon EDT
Clarification Requests Accepted	Thu., April 16	12:00 Noon EDT
Clarifications Posted	Thu., April 16	4:00 pm EDT
Submissions Due:	Fri., April 17	12:00 Noon EDT

Instructions

You are to work on this exam wherever you are. The following instructions give very detailed guidelines on what resources you can reference while working on the exam and any communications you can have regarding the contents of the exam.

As noted above, we will accept requests for clarifications and for additional resources during the first 24 hours of this exam. These should be posted as private messages on the class Piazza site. We will respond to these requests by either making Piazza posts (to the entire class if we believe clarifications are necessary), or by adding additional links to the Exam 2 resource web page:

https://www.cs.cmu.edu/˜418/exams/resources-exam2/resources.html

We will attempt to respond as soon as possible, but in any case, all clarifications and additional resources will be available by 4:00 pm on Thu., April 16. After that, in fairness to the people working across many time zones, we will not provide any further information about the exam. Be sure to check Piazza and to get the final version of the exam once all clarifications have been posted. We will not be holding office hours during the exam period. All communication to the teaching staff should be via posts to Piazza.

Some Advice: We recommend you attempt to do as much of the exam as you can during the first 24 hours so that you will be able to make effective requests for clarifications and information. Use the final 20 hours to make sure your answers are consistent with any additional information provided.

We have limited ability to verify that you have followed these guidelines. We are therefore relying on an honor system. We expect that you will follow these rules out of your own sense of honesty and integrity to yourself, to your education, to Carnegie Mellon University, and to the other students. We will require you to signify that you have followed our guidelines with a pledge when you submit your solution.

You will submit an electronic version of your solution to Gradescope as a PDF file. We have prepared a separate document for you to provide your answers, available at:

https://www.cs.cmu.edu/˜418/exams/exam2-answer.pdf

For those of you familiar with the LaTeX text formatter, you can download the template file at:

https://www.cs.cmu.edu/˜418/exams/exam2-answer.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use the PDF document as your starting point. You can either: 1) electronically modify the PDF, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of the answer document.

Allowed Resources

As a general rule, you are allowed to view a document from the WWW *only if* it has a URL beginning with one of the following prefixes:

- `https://www.cs.cmu.edu/~418/`
- `https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15418-s20/www/`
- `https://piazza.com/class/k4ylja6hsj86xn` (The class Piazza site)
- `https://www.gradescope.com/courses/78440` (The class Gradescope site)

These include the following:

- All course information pages
- All handouts, including slides, assignments, exercises, exercise solutions, practice exams, and practice exam solutions.
- All code that has been provided to you as examples
- The information pages we provide at the Exam 2 resource web page.

In addition you may make use of the following:

- Any videos that have been linked from the course schedule web page
- Any code we provided to you for one of the assignments or for any of the code you developed for an assignment.
- Any notes you have prepared during the course or in preparation for the exam.
- You may write and test programs in any language you choose.
- You may use calculators, spreadsheets, and other productivity tools.

The above set of resources is exclusive. If there are additional resources you believe would be useful and appropriate, you may post a request and we will consider whether to make them available. Examples could include:

- Wikipedia entries

- Linux man pages
- Documentation on some language feature that you would like to use.

We will provide these resources by making a copy of the desired web document, stripping out any hyperlinks, and providing them at the Exam 2 resource web page.

Disallowed Resources

Do not trust your own judgment on what additional resources are appropriate for you to use. Only those we have explicitly allowed can be used.

You may not follow links accessible from the class web page, if they do not begin with one of the prefixes listed above. In particular, you may not use any resources from previous versions of this course.

You may not make use of any additional material you may have on hand, such as textbooks or notes from other courses. If there is additional background material that you think is necessary, you should post a request. Note, however, that we expect you to have competencies in C/C++ programming, basic data structures, and basic math.

Allowed Communications

You may not have any form of communication, electronic or otherwise, with anyone except the course instructors and teaching assistants regarding the contents of this exam. As described above, we will accept requests for clarifications or for additional resources during the first 24 hours of the exam.

Problem 1: Multiple Choice (15 points)

Problems that state “list all that apply” may have any number of correct answers (including zero). If none are correct, list “none.” The other problems have unique answers.

A. Which of the following memory consistency policies allow the following behavior? Assume x and y start with value zero. (List all that apply.)

```
Thread 0       Thread 1
--------       --------
x <- 1         y <- 1
print(y)       print(x)

Output:
0
0
```

(a) Non-coherent shared memory.

(b) Coherent shared memory.

(c) Total-store-order (TSO) consistency.

(d) Sequential consistency.

B. What is the key optimization enabled by total-store-order (TSO) consistency vs. sequential consistency (SC)? (List all that apply.)

(a) MESI coherence.

(b) Prefetching.

(c) Atomic memory operations.

(d) Store buffers.

C. Which of the following statements are true about implementations of MPI (list all that apply)

(a) MPI assumes the program is running on a machine with a cache-coherent shared address space.

(b) The MPI library allocates the needed space for all buffers.

(c) Synchronous communication creates a synchronization barrier for the sender and receiver.

(d) A sender can modify the send buffer as soon as the call to `MPI_Send` returns.

D. Which of the following statements are true about implementations of OpenMP (list all that apply)

(a) OpenMP assumes the program is running on a machine with a cache-coherent shared address space.

(b) OpenMP can only exploit parallelism by splitting up the index ranges in `for` loops.

(c) OpenMP implements all atomic operations via locks.

(d) OpenMP reduction assumes the reduction operation is associative.

E. Assume that in a computer system, all operations continually abort and retry. Which one of the following could describe this scenario?

(a) Deadlock.

(b) Livelock.

(c) Starvation.

(d) None of the above.

F. To implement a lock-free single reader, single writer unbounded queue, how many additional pointers does the program need beyond those needed for a sequential implementation?

(a) 0

(b) 1

(c) 2

(d) 3

G. Which of the following statements is/are true? List all that apply.

(a) Test & Test and Set locks guarantee fairness of locks, as opposed to Test and Set locks

(b) It is better to spinlock than block a thread if scheduling overhead is larger than expected wait time

(c) You can replace all uses of locks with atomic transaction regions

(d) Pessimistic conflict detection detects conflicts earlier than optimistic detection

H. Which of the following are domain-specific languages? List all that apply.

(a) Ruby.

(b) Liszt.

(c) Halide.

(d) C++.

I. You are tasked with implementing a hardware transactional memory system in a new multicore processor with a local cache for each processor. Which of the following pieces of information are necessary for you to proceed with your implementation? (List all that apply)

(a) The cache coherence protocol

(b) The clock rate of each core

(c) The number of cores

(d) The versioning policy

J. When training neural networks on a cluster and using a parameter server, what can we always expect?

(a) Parameters are always fresh

(b) The neural network is running in some model-parallel manner

(c) Training will always converge

(d) Updates to the server happen asynchronously

K. Which of the following statements is/are true? (list all that apply)

(a) ASICs have longer development workflows than FPGAs

(b) An ASIC circuit is more reconfigurable than a FPGA circuit

(c) ASICs developed for a given task are faster than FPGAs at that task

(d) ASICs developed for a given task use more power than FPGAs for the same task

L.

```
lock:      ts  R3, memory[address]
           bnz R3, lock
unlock :   st  memory[address], #0
```

Assume the above assembly code follows a RISC ISA, with **ts** representing the test and set instruction. Assume that four threads running on four different processors are trying to acquire the lock. Which of the following is a/are plausible scenarios due to given lock mechanism?

(a) Starvation

(b) Deadlock

(c) Above lock doesn't ensure mutual exclusion

(d) All of the above

M. Which of the following performance evaluation tools has the least overhead?

(a) valgrind
(b) gprof
(c) top
(d) pin

N. What is the worst case latency of an n-node mesh network?

(a) $O(\sqrt{n})$
(b) $O(\log n)$
(c) $O(n)$
(d) $O(n\ \log n)$

O. Which of the following interconnect networks is non-blocking?

(a) Mesh
(b) Fat Tree
(c) Tree
(d) Multi-stage Logarithmic

```
/* Data type for ticket values */
typedef unsigned long ticket_value_t;

/* Data structure for ticket lock */
typedef struct {
    ticket_value_t next_ticket;
    /* Padding to force other field into a separate cache block */
    uint64_t padding[8];
    volatile ticket_value_t serving;
} ticket_lock_t;

/* Initialize lock */
void ticket_lock_init(ticket_lock_t *lock) {
    lock->next_ticket = 0;
    lock->serving  = 0;
}

/* Acquire lock */
void ticket_lock_lock(ticket_lock_t *lock) {
    ticket_value_t my_ticket = __sync_fetch_and_add(&lock->next_ticket, 1);
    /* Busy Wait */
    while (1) {
        ticket_value_t current_ticket = lock->serving;
        if (current_ticket == my_ticket)
            break;
    }
}

/* Release lock */
void ticket_lock_unlock(ticket_lock_t *lock) {
    lock->serving ++;
}
```

Figure 1: Ticket lock implementation

Problem 2: Locks and memory consistency (18 points)

Figure 1 shows code implementing a ticket lock, a form of mutual exclusion lock. It makes use of an atomic fetch-and-add, provided as a GCC builtin. These were covered briefly in Lecture 16.

2A: Lock properties (12 points)

In answering the following questions, you may assume the following:

- The main program using the lock will have the structure of the code shown in Figure 2.
- The underlying memory system obeys sequential consistency

```
ticket_lock_t mutex;
/*
  Other global data declarations
*/

/* Thread routine */
void thread_fun(void *vargp) {
   long niters_per_thread = (long) vargp;

    for int i = 0; i < niters_per_thread; i++) {
        /*
        ... Read and write local data
        */
        ticket_lock_lock(&mutex);
        /*
        Critical Section
        ... Read and write global data
        */
        ticket_lock_unlock(&mutex);
    }
    return NULL;
}


int main(int argc, int argv) {
    int nthreads = atoi(argv[1]);
    int niters = atoi(argv[2]);
    long niters_per_thread = niters / nthreads;
    pthread_t tid[nthreads];

    ticket_lock_init(&mutex);

    /* Create peer threads and wait for them to finish */
    for (int t = 0; t < nthreads; t++)
        pthread_create(&tid[t], NULL, thread_fun, (void *) niters_per_thread);

    for (int t = 0; t < nthreads; t++)
        pthread_join(tid[t], NULL);

    return 0;
}
```

Figure 2: Structure of the main program making use of ticket lock

- The fetch-and-add operation is atomic and always completes.

Provide brief answers to the following questions:

(1) The C keyword `volatile` indicates to the compiler that changes to a value may occur between different accesses, even if it does not appear to be modified. Why must the field `serving` in `ticket_lock_t` be declared as volatile to guarantee correct compilation?

(2) Does this lock guarantee fairness (i.e., any request will eventually be granted)? Explain.

(3) Why can the incrementing of `lock->serving` in function `ticket_lock_unlock` be done with ordinary addition rather than atomic fetch-and-add?

(4) We saw in Exercise 4 that we could improve the performance of a lock based on atomic compare-and-swap by inserting a random delay in the busy-waiting loop, with the amount of delay following an exponential distribution. In similar experiments for a ticket lock, this scheme leads to worse performance. Explain why that could be the case.

(5) Suppose the definition of type `ticket_value_t` were changed from `unsigned long` to `unsigned char`. Would this impose any additional limits on the values of `niters` or `nthreads` in the main program? Explain.

(6) Suppose you run the main program of Figure 2 on an 8-core processor with `nthreads` set to 8. The machine uses a bus-based cache coherence system following an MSI protocol. Assume at some point in the main program execution that Thread 0 is in its critical section and all other threads are in their busy-wait loops, attempting to acquire the lock, with Thread 1 having the smallest ticket value and Thread 7 having the largest. For all of the caches, describe which field(s) of the lock would be held in each cache and what their state(s) would be.

(7) Describe all of the bus traffic and cache transitions that would occur due to the following sequence. Describe only the traffic relevant to the lock. Assume there are no conflict misses. For each of the actions listed, describe any resulting actions that would occur by other threads or caches until a steady state is reached.

  (a) Thread 0 exits its critical section and releases the lock, and Thread 1 then acquires the lock and enters its critical section.

  (b) Thread 0 attempts to acquire the lock and re-enters its busy-wait loop.

```
/* Initialize lock */
void ticket_lock_init(ticket_lock_t *lock) {
    lock->next_ticket = 0;
    lock->serving  =0;
    // Possible fence #1
}

/* Acquire lock */
void ticket_lock_lock(ticket_lock_t *lock) {
    ticket_value_t my_ticket = __sync_fetch_and_add(&lock->next_ticket, 1);
    /* Busy Wait */
    while (1) {
        // Possible fence #2
        ticket_value_t current_ticket = lock->serving;
        // Possible fence #3
        if (current_ticket == my_ticket)
            break;
    }
    // Possible fence #4
}

/* Release lock */
void ticket_lock_unlock(ticket_lock_t *lock) {
    // Possible fence #5
    lock->serving ++;
    // Possible fence #6
}
```

Figure 3: Ticket lock implementation with possible locations for fence operations

2B: Memory consistency (6 points)

In answering the following questions, you may assume the following:

- The program using the lock will have the structure of the main program shown in Figure 2.
- The underlying memory system provides only the following consistency guarantees:
  - Within a thread, all sequential ordering constraints are maintained.
  - For any single memory location, all loads and stores of that location are sequentially consistent.

  Specifically, there are no guarantees about the ordering of loads and stores made by different threads to different memory locations.
- The fetch-and-add operation is atomic and always completes.
- The fetch-and-add operation obeys the following property, described in the GCC builtin documentation:

> . . . these builtins are considered a *full barrier*. That is, no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

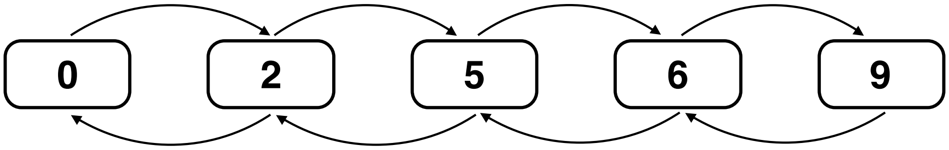
- There is a library function `fence` that also serves as a full barrier.

Figure 3 shows the ticket lock code with six locations where it may be appropriate to insert a fence operation. We want to insert a minimum number of fence operations such that correct synchronization of the main program is guaranteed. Select an appropriate set of these. For each of the six possible fence locations either: 1) justify why it is required by describing a scenario in which incorrect behavior could arise without it, or 2) justify why it is not required.

Problem 3: Managing Concurrrency (10 points)

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_from_head`, which traverses the list from the head. The implementation uses hand-over-hand locking by adapting the code shown in Slide 27 of Lecture 18.
- `delete_from_head`, which deletes a node by traversing from the head, using hand-over-hand locking shown on the same slide.
- `insert_from_tail`, which traverses the list **backwards from the tail** to insert a node using hand-over-hand locking in the opposite order as `insert_from_head`



A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

- Test 1: `insert_from_head(1), delete_from_head(9)`
- Test 2: `insert_from_head(8), delete_from_head(2)`
- Test 3: `insert_from_head(8), insert_from_tail(1)`

The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

B. Imagine you removed all locks and implemented `insert_from_head`, `delete_from_head`, and `insert_from_tail` by placing the entire body of these functions in an atomic block for execution on a system **supporting lazy optimistic transactional memory with aggressive conflict resolution**, as described in Lecture 19 Does this fix the correctness problem described in part A? Why or why not?

C. Consider two transactions simultaneously performing `insert_from_head(3)` and `delete_from_head(9)`. Assume both transactions start at the same time on different cores and the transaction for `insert_from_head(3)` proceeds to commit while the `delete_from_head(9)` transaction has just iterated to the node with value 6. Must either of the two transactions abort in this situation? Why? **(Remember this is a lazy optimistic transactional memory system!)**

D. Must either transaction abort if the transaction for `delete_from_head(9)` proceeds to commit before the transaction for `insert_from_head(3)` does? Why? **Please assume that at the time of the attempted commit, `insert_from_head(3)` has iterated to node 2, but has not begun to modify the list.**

E. Must either transaction abort if the situation in part D is changed so that `delete_from_head(9)` attempts to commit first, but by this time `insert_from_head(3)` has made updates to the list (although not yet initiated its commit)? Why?

Problem 4: Homogeneous vs. Heterogeneous Parallelism (20 points)

This problem concerns the costs and benefits of heterogeneous parallelism.

We will begin by considering multicore CPU processors, comparing the performance of symmetric (homogeneous) and asymmetric (heterogeneous) designs.

Imagine you are a hardware architect designing a new multicore CPU. Suppose that the CPU can take at most N resources (e.g., resources could be area of at most 200 mm^2 or power of at most 100 W).

You can design CPU cores with the following characteristics:

- **Out-of-order ("OOO"):** The OOO core achieves sequential speedup (vs. some baseline core) as a function of resources r
$$\text{speedup}_O(r) = \sqrt{r+1} - 1 \tag{1}$$
- **In-order ("IO"):** The IO core achieves sequential speedup (vs. some baseline core) as a function of resources r:
$$\text{speedup}_I(r) = 1 - \frac{1}{2r+1} \tag{2}$$

For this problem, all speedups are normalized to a core with sequential performance $= 1$. Feel free to give approximate numerical answers (e.g., within 0.1 of optimal), and we recommend that you write a simple program or use spreadsheet software to get numerical answers.

4A: Homogeneous multicore (7 points): Suppose an application spends a fraction f of its execution running in parallel with ideal speedup, and $1 - f$ executing sequentially.

(1) What is the speedup **in the sequential region** of the program for a single core of size r? (Give an expression involving N, f, and r.)

OOO Core.

IO Core.

(2) What is the speedup **in the parallel region** of the program for a homogeneous multicore? (Give an expression involving N, f, and r.)

OOO Core.

IO Core.

(3) What is the **end-to-end speedup** for a homogeneous multicore? (Give an expression involving N, f, and r.)

OOO Core.

IO Core.

(4) Suppose that $f = 0.9$ and $N = 16$. What speedup is achieved by the **best** homogeneous OOO and IO designs? Which performs better? How many cores does the best design have?

(5) Now suppose that $f = 0.8$ and $N = 16$. What speedup is achieved by the **best** homogeneous OOO and IO designs? Which performs better? How many cores does the best design have?

(6) Compare your last two answers for $f = 0.9$ to $f = 0.8$. What do they say about how applications affect the best architecture? About the resilience of different architectural choices to different applications?

4B: Heterogeneous multicore (8 points): Now suppose that you are able to combine different kinds of cores in your multicore design. Let's consider the performance of a heterogeneous multicore CPU using a mix of OOO and IO cores, written as a function of f, N, r_O (out-of-order core size) and r_I (in-order core size).

Note: The optimal heterogeneous design will consist of a single OOO core plus many IO cores. The sequential region runs on the OOO core.

(1) What is the speedup **in the sequential region** of the program? (Give an expression involving N, f, r_O, and r_I.)

(2) What is the speedup **in the parallel region** of the program for a heterogeneous multicore? (Give an expression involving N, f, r_O, and r_I.)

(3) What is the **end-to-end speedup** for a heterogeneous multicore? (Give an expression involving N, f, r_O, and r_I.)

(4) Suppose that $f = 0.9$ and $N = 16$ and, for now, fix in-order cores at size $r_I = 1$. What speedup does the **best** heterogeneous design achieve?

(5) What is the **best** overall speedup when the **in-order core size** r_I **varies** from 0.1, 0.5, 1.0, 2.0, to 4.0?

(6) Qualitatively, what does this say about the optimal heterogeneous architecture? (Does it remind you of any designs you've used this semester?)

4C: Scaling (5 points): Finally, let's use the model you've already developed to see how things change with different resource budgets. For the rest of this problem, fix $f = 0.9$ and $r_I = 0.1$ in the heterogeneous design.

(1) What are the best **heterogeneous** and **homogeneous** designs when $N = 4$?

(2) What are the best **heterogeneous** and **homogeneous** designs when $N = 64$?

(3) What does this say about the benefits of heterogeneity as resource budgets increase over time?

```
// Process input data streams with convolutions to generate output data streams
void convolve(int dsize, int input_depth, int output_depth, int csize,
              float input[dsize+csize-1][input_depth],
              float output[dsize][output_depth],
              float weights[output_depth][csize][input_depth]) {
    for (int i = 0; i < dsize; i++) {
        for (int dout = 0; dout < output_depth; dout++) {
            float tmp = 0.0;
            // Convolution operation for single output value
            for (int ii = 0; ii < csize; ii++) {
                for (int din = 0; din < input_depth; din++) {
                    tmp += weights[dout][ii][din] * input[i+ii][din];
                }
            }
            output[i][dout] = tmp;
        }
    }
}
```

Figure 4: Convolution computation

Problem 5: Memory Performance during Convolution Computation (18 points)

This problem concerns the performance of a program performing a *convolution* operation over a set of data, such as occurs in deep neural network evaluation. It is similar to the example shown in Slide 22 of Lecture 24, but simplified to operate on one-dimensional data.

Computation

Figure 4 shows code to implement one-dimensional convolution. Function `convolve` is given four size parameters:

- Data size `dsize` N. The input and output data streams have length N.
- Input depth `input_depth` D_i. Each input data stream consists of D_i elements
- Output depth `output_depth` D_o. Each output data stream consists of D_o elements
- Convolution size `csize` K. Each filter performs a $K \times D_i$ convolution to compute each output data element.

The function generates $N \times D_o$ output data values based on $N \times D_i$ input data values using a set of $K \times D_i$ convolutions.

Note that the input stream is sized to have length $N + K - 1$ to avoid out-of-bounds array references. In practice, these additional $(K - 1) \times D_i$ values are generated by replicating streams at each end of the data.

```
#define DSIZE 20000
#define CSIZE 17
#define IN_DEPTH 192
#define OUT_DEPTH 192
float input[DSIZE+CSIZE-1][IN_DEPTH];
float output[DSIZE+CSIZE-1][OUT_DEPTH];
float weights[OUT_DEPTH][CSIZE][IN_DEPTH];

void stage() {
    convolve(DSIZE, IN_DEPTH, OUT_DEPTH, CSIZE,
             input, output, weights);
}
```

Figure 5: Parameters for convolution benchmark

We see that the function contains four nested loops. The outer two select which particular output data element to compute. The inner two perform the convolution as the weighted sum of $K \times D_i$ input data elements. Note that there is a separate weight for each output element, but the set of weights is independent of the data index `i`.

Figure 5 shows a particular set of parameters for the `convolve` operation that we will use in our performance analysis, with $N =$20,000, $D_i = D_o =$ 192, and $K =$ 17.

Machine Model

In order to estimate the performance of this function we will use the following, highly stylized model of machine performance. We assume the following:

- The processor has 8 cores, each with an L1 cache able to hold 32,768 bytes (8,192 floats.)
- There is a shared L2 cache, able to hold 1,048,576 bytes (262,144 floats.)
- All caches have block sizes of 16 bytes (able to hold 4 floats).
- The caches have enough associativity that we can ignore conflict misses, and they use an LRU replacement policy.
- The time for a load operation and its effect depend on the memory level from which it must be retrieved:

  L1: The value will be available to the CPU in 1 clock cycle.

  L2: The value will be available to the CPU in 9 clock cycles. It will also be loaded into the L1 cache.

  Memory: The value will be available to the CPU in 49 clock cycles. It will also be loaded into the L1 and L2 caches.
- We can predict the running time of a program based only on the time it spends performing load operations.

- The processor performs loads in program order, and it does not implement any memory latency-hiding techniques, such as load speculation or prefetching.

Given all of this, we will express the performance of a core running the code in units of *cycles per inner step* (CPIS), defined as the average number of clock cycles required by the load operations in the inner loop of a function. For the case of `convolve`, there are two loads: one for a weight and one an input value. If both values were always in the L1 cache, the CPIS would be 2.0. If both were always in memory, the CPIS would be 98.0. The predicted performance will lie somewhere between these two extremes.

5A: Sequential performance (8 points)

For all of questions requiring you to express performance in CPIS, you need only give it to the first decimal position (e.g., 3.7). Therefore, you can make many simplifying assumptions in your performance estimates. Don't waste your time fussing over low-level details. In particular, you can often consider the typical behavior that would occur during the middle of a loop and ignore effects that occur only at the beginning or the end of a loop.

A. Consider the following code:

```
#define LEN 5932
float data[LEN];
float sum = 0.0;
for (int i = 0; i < LEN; i++)
   sum += data[i];
```

Calculate the CPIS for this code assuming array `data` is initially in the following memory levels. Remember: report these numbers to the first decimal place.

L1 cache:

L2 cache:

Main memory:

B. Determine the following regarding function `convolve`:

(1) The total number of input data elements accessed during the computation, and the highest memory level in which this would fit (where L1 is highest and main memory is lowest.)

(2) The number of input data elements accessed for one value of outer loop index `i`, and the highest memory level in which this would fit.

(3) The number of times each input data element is accessed for one value of outer loop index `i`.

C. Determine the following regarding function `convolve`:

(1) The total number of weight values accessed during the computation, and the highest memory level in which this would fit.

(2) The number of weight values accessed for one value of outer loop index `i`, and the highest memory level in which this would fit.

```
// Process input data with convolutions to generate output data
void convolve_par(int dsize, int input_depth, int output_depth, int csize,
                  float input[dsize+csize-1][input_depth],
                  float output[dsize][output_depth],
                  float weights[output_depth][csize][input_depth]) {
    #pragma omp parallel for
    for (int i = 0; i < dsize; i++) {
        int thread_count = omp_get_num_threads();
        int thread_id = omp_get_thread_num();
        for (int dout = 0; dout < output_depth; dout++) {
            float tmp = 0.0;
            // Convolution operation for single output value
            for (int ii = 0; ii < csize; ii++) {
                // Restructure this inner loop to improve cache performance
                for (int din = 0; din < input_depth; din++) {
                    tmp += weights[dout][ii][din] * input[i+ii][din];
                }
            }
            output[i][dout] = tmp;
        }
    }
}
```

Figure 6: Parallel convolution computation

(3) The number of times each weight value is accessed for one value of outer loop index `i`.

D. Determine the single-core performance of function `convolve`:

(1) Describe the access pattern for the input data within the innermost two loops, in terms of which levels of memory the data would reside in.

(2) What would be the resulting average number of cycles per load operation for the input data?

(3) Describe the access pattern for the weights within the innermost two loops, in terms of which levels of memory the weights would reside in.

(4) What would be the resulting average number of cycles per load operation for the weights?

(5) What is the CPIS for the function?

5B: Parallel performance (10 points)

Figure 6 shows a modified version of the `convolve` function designed for parallel execution. It contains an OpenMP pragma to apply static parallelism to the outermost loop.

To simplify the performance analysis, assume that the memory operations across threads occur in *lock step*. That is, all threads would initiate the load operations on their respective values of the `weights` array, and then they would all initiate the load operations on their respective values of the `input` array. In both cases, the time required by the loads would be determined by that of the slowest thread.

A. Determine the multicore performance of function `convolve_par` when running with T threads for $T \leq 8$.

   (1) For a given thread, describe the access pattern for the input data within the innermost two loops, in terms of which levels of memory the data would reside in.

   (2) For a given thread, what would be the resulting average number of cycles per load operation for the input data?

   (3) For all threads, describe the access pattern for the weights within the innermost two loops, in terms of which levels of memory the weights would reside in.

   (4) For a given thread, what would be the resulting average number of cycles per load operation for the weights?

   (5) What is the CPIS per thread for the function?

   (6) What is the speedup, relative to sequential execution, as a function of T?

B. Describe how you could restructure the order in which elements are accessed within the inner loop of the code shown in Figure 6 to give the program superlinear speedup. You can assume that $T \in \{1, 2, 4, 8\}$. You may wish to make use of the values `thread_count` and `thread_id` that are computed within the parallel block. You should write code for the restructured loop.

C. What would be the resulting CPIS per thread for $T \in \{2, 4, 8\}$? Explain.

D. What would be the speedup, relative to the sequential code for $T \in \{2, 4, 8\}$?

Pledge

On the answer sheet, you must (electronically) sign the following pledge:

> I do hereby swear that, in generating my answers to this exam, I made use of only the resources explicitly listed in the exam document. I did not have any communication of any form with anyone other than the course instructors and teaching assistants about the contents of this exam.
>
> Signed