Full Name: ______________________________

SUNet Id: ______________________________

I accept the letter and spirit of the honor code:

Signed: ______________________________

Stanford CS149, Fall 2022

Final Exam Solutions

Dec 15th, 2022

Instructions:

- Write your answers in the space provided below the problem. If your work gets messy, please clearly indicate your final answer.
- The exam has a maximum score of **100** points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.

Problem	Your Score	Possible Points
1		24
2		14
3		16
4		14
5		16
6		18
Total		100

Miscellaneous Short Problems

Problem 1. (24 points):

A. (4 pts) After finishing CS149, you decide to join a research team that is building a new multi-core processor. At the first design meeting, you learn the team is building an **eight-core processor with a single 8 MB cache that is shared among all cores**. If the team wants a coherent memory system, should the team spend time designing a cache coherence protocol for the multi-core chip, why or why not?

Solution: Since there is only a single cache, there are not multiple caches to keep coherent. In other words, there is no need to have a coherence protocol that prevents different values for an address being stored in different caches, because the processor does not have more than one cache!

B. (4 pts) Your friend is designing a soccer playing robot for the next RoboCup competition. The software on the robot must send a torque request to the robot's motors every 1 ms in order for the robot to successfully locomote. Unfortunately in your friend's implementation, computing torques takes 10 ms when running serially on a single core of the robot's 1 GHz single-core CPU. As a result your friend's robot constantly falls over without being touched. You laugh at your friend, and call their robot "Neymar"!

You look at your friend's code and notice that 20% of the code is inherently serial. The rest is perfectly parallelizable. You dig into your desk drawer and find two processors: Processor A is a 32-core processor that runs at 1 GHz. Processor B is a 8-core processor that runs at 4 GHz. Can you solve your friend's performance problem? Justify your answer by computing the maximum performance they can achieve.

Solution: They can meet their performance goal using processor B. This processor can compute the serial portion is 2ms/4 = .5 ms and complete the parallel portion in 8ms/8/4 = .25 ms, so the whole computation runs in .75 ms. With processor A, the serial portion of the code alone takes 2 ms.

C. (4 pts) When we discussed Cilk, we emphasized how `cilk_spawn foo()` differs from a normal C function call `foo()` in that the Cilk call can run asynchronously with the caller. Notice that Cilk doesn't explicitly state that the callee function runs **in parallel with the caller**. Give one reason why the designers of Cilk intentionally designed a language that does not specify when the call will run relative to the caller?

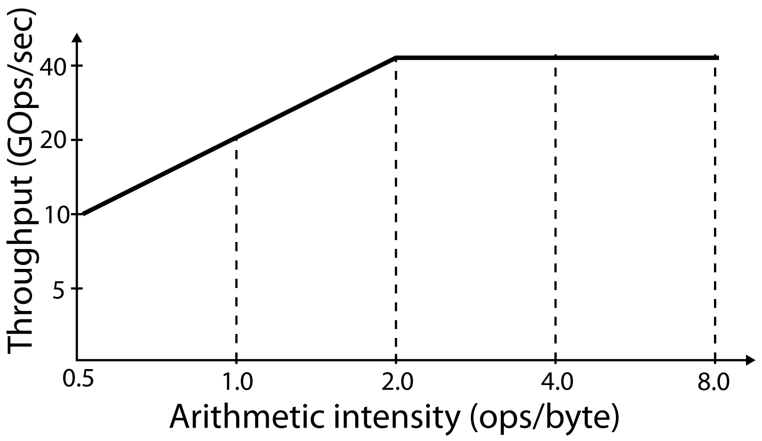
Solution: Flexibility in scheduling allows the runtime to choose a good schedule policy for the machine or workload at hand. By keep scheduling out of the code, implementations can achieve performance portability by using different scheduling strategies on different machines.

D. (4 pts) What is the peak arithmetic throughput (in flops/clock) of a 10 core processor, where each core supports 8-way hardware multi-threading (interleaved multi-threading as discussed in class) and is capable of executing up to two 4-wide SIMD floating-point instructions per clock in a superscalar fashion?

Solution: 10 cores $\times$ 2 instr $\times$ 4 ops = 80 floating point ops per clock. Note that the number of threads per core is not relevant to a computation of the processor's peak throughput.

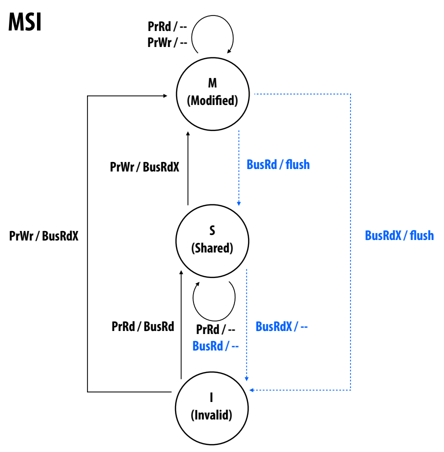
E. (4 pts) The figure below shows a "roofline" plot for a specific processor. Each point on the graph corresponds to the performance (the Y axis is ops/sec) of a different program with the given arithmetic intensity (the X-axis gives the arithmetic ops per byte read by the program. **What is the bandwidth available to the processor? Please give a number, and a 1-2 sentence explanation of your answer.**

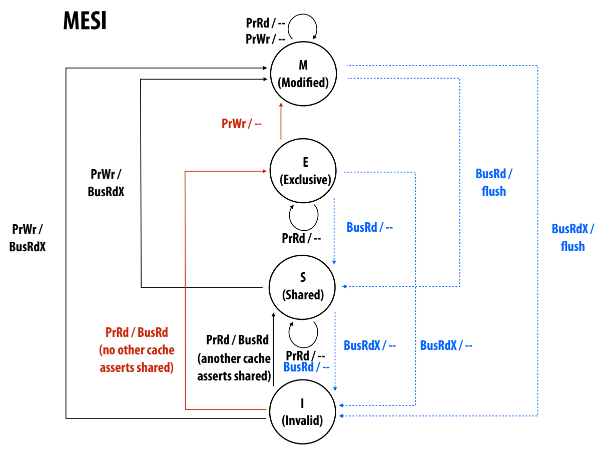
Hint: consider why the curve slants diagonally upward in the left part of the graph. Why does the curve flatten out on the right side?



Solution: 20 GB/sec. Notice that when the problem is bandwidth bound, the processor performs 20 GOps/sec when the program have arithmetic intensity 1. Therefore it must be reading 1 byte for each of these ops, or reading data from memory at 20 GB/sec. You could derive this number of any of the bandwidth-bound data points, or the data point at the "knee" of the curve, which is the point the processor transitions from being BW-bound to compute bound.

F. (4 pts) We have used the MSI cache coherence protocol for many cache coherence problems, but after Argentina's run to the World Cup final, many students asked for a question about the MESI protocol. Below are the cache coherence diagrams for MSI and MESI. Note the major difference is the introduction of the "E" state, which represents "exclusive clean". In other words, a cache with a line in the "E" state is the only cache holding the line, but the cache only has permission to read the line. Notice in the coherence diagram that moving a line from the E to the M state in MESI generates no bus traffic notifying the other processors. Please (1) describe why no coherence messages need to be sent during a E-to-M transition, and (2) **please give a minimal sequence of pseudocode** for a sequence of memory operations where the MESI protocol would be more efficient (in terms of the amount of cache coherence traffic required than the MSI protocol.





Solution: When transitioning from E to M, it is not possible for another cache to have the line (by the definition of the E state). As a result, the cache transitioning to M can safely assume it has exclusive access to the line. Consider the sequence...

load X store X
In the MSI protocol, this will generate a `BusRd` *followed by a* `BusRdX`*. In MESI, all processes see the* `BusRd`*, but don't need to be further notified when the processor "upgrades" the line from the E to the M state.*

Understanding the behavior of caches!

Problem 2. (14 points):

Consider the following C code run by one thread on a CPU featuring a cache with 64 byte (16 float) cache lines and a total capacity of 128 KB. Assume the cache employs a least recently used (LRU) eviction policy and it is fully associative (any cache line can be placed in any available location in the cache). **The processor does not perform any data pre-fetching.**

```
const int SIZE = 1024 * 1024 * 1024;
float input[SIZE];          // sizeof(float) * SIZE bytes
float output[SIZE];         // sizeof(float) * SIZE bytes

// assume input is appropriately initialized here

for (int i=0; i<SIZE; i++) {
  output[i] = input[i] * 2.0;
}
```

A. (4 pts) In the `for` loop, what fraction of loads from `input` are cache misses? Briefly explain why? **(Hint: Please keep in mind the cache line size.)**

Solution: 1 of every 16 of loads are misses. After each cold miss, there are 15 consecutive cache hits since subsequent accesses to the next elements of `input` lie on the recently loaded line. The program has high spatial locality.

B. (4 pts) Imagine the code is changed in the following way to have a nested set of `for` loops:

```
const int SIZE = 8 * 1024;
float input[SIZE];     // sizeof(float) * SIZE bytes
float output[SIZE];    // sizeof(float) * SIZE bytes

// assume input is appropriately initialized here

for (int j=0; j<10000; j++) {
  for (int i=0; i<SIZE; i++) {
    output[i] = output[i] + input[i] * 2.0;
  }
}
```

Assume the processor's cache is the same as in part A. In the "common case" where `j>0`, what fraction of accesses to `input` and `output` (both loads and stores) are cache misses? Briefly explain why?

Solution: All loads are hits since both the `input` and `output` arrays fits in the cache. Each array is 32 KB, and the cache is 128 KB.

C. (6 pts) Now consider the following code which works on 2D arrays:

```
const int SIZE_1 = 1024 * 1024 * 1024;
const int SIZE_2 = 1024 * 8;

float input[SIZE_1][SIZE_2];
float output[SIZE_1-1][SIZE_2];

// assume input is appropriately initialized here

for (int j=0; j<SIZE_1-1; j++) {
  for (int i=0; i<SIZE_2; i++) {
    output[j][i] = input[j+1][i] + input[j][i];
  }
}
```

Now assume that the cache still has the 64 byte line size as before, but **is now only 32 KB in size**. Please rewrite the code so that you minimize the number of cache misses. You can only modify the loop structure (you can add/remove loops and adjust loop bounds) and modify array indexing, but you cannot change the number of mathematical operations performed. **Pseudocode is fine, it need not compile. But to help the grader understand your code, please briefly (in a sentence or two) describe the rough approach used in your solution.**

Solution: block the `i` loop into chunks. To maximize spatial locality, the value of `CHUNK` should be a multiple of 64/`sizeof(float)`=16 array elements. To maximize temporal locality, `CHUNK` should be small enough that the prior row of `input` remains in cache when accessed in the next iteration of the `j` loop.

We were fine if students ignored the fact that `output` also needs to be stored in the cache, and simply computed that the value of `CHUNK` needs to be small enough so that two rows of `input` can fit in cache. In this case, that would be rows of 4K elements, so `CHUNK <= 4 * 1024`.

```
for (int ii=0; ii<SIZE_2; ii+=CHUNK) {
  for (int j=0; j<SIZE_1-1; j++) {
    for (int i=0; i<CHUNK; i++) {
      output[j][ii+i] = input[j][ii+i] + input[j-1][ii+i];
    }
  }
}
```

Problem 3. (16 points):

English football superstar Harry Kane starts preparing for the next World Cup by developing an algorithm for simulating penalty kicks. The algorithm simulates many penalty kicks in parallel by running a function `simulate_random_kick()` in each thread. When complete, the program prints out the height of the lowest kick found so that Harry has a target in mind for future practice sessions.

The code below uses the function `atomicCAS`, which was discussed in class. **The definition of `atomicCAS` is given for convenience, but please keep in mind this is an atomic operation that is treated as a write by the cache coherence protocol.**

```
// REMEMBER THIS IS EXECUTED ATOMICALLY
int atomicCAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
int lowest_so_far;    // shared global variable among threads,
                      // holds lowest height so far in centimeters

void thread_main() {    // assume this code is run by many threads

   float ball_height;

   for (int i=0; i<1000000; i++) {
     simulate_random_kick(&ball_height);      // runs a sim that fills in ball height
     int old_lowest = lowest_so_far;
     int min_height = min(ball_height, old_lowest);
     while (atomicCAS(&lowest_so_far, old_lowest, min_height) != old_lowest) {
       old_lowest = lowest_so_far;
       min_height = min(ball_height, old_lowest);
     }
   }

   barrier();      // a regular barrier across all threads

   if (get_thread_id() == 0) {  // assume get_thread_id() behaves as expected
      printf("The lowest height is %d cm\n", lowest_so_far);
   }
}
```

A. (2 pts) In your own words, describe the correctness reason for why there is a barrier in this program. (How might the program give incorrect results if there is no barrier?)

Solution: Need to make sure all threads have completed their work prior to printing out the lowest kick. Otherwise thread 0 might print results prior to all other threads finishing their simulations.

B. (4 pts) Imagine the case where **`simulate_random_kick()` is a very expensive function that takes over a second to compute. All calls to `simulate_random_kick()` take the same amount of time**. Please describe whether you think this program will achieve near perfect speedup on a multi-core processor that implements invalidation-based cache coherence. Please describe why or why not? If you think there is a way to improve its speedup substantially, describe how you might improve the code (rough pseudocode is fine).

Solution: We'd expect good code speedup (very close to linear). The run time is dominated by the call to `simulate_random_kick`, and equal work is done by all processors. The cost of the `CAS` synchronization and the `barrier` should be minimal.

C. (4 pts). Now imagine the case where **`simulate_random_kick()` is a very lightweight function that takes just a few instructions. All calls to the function take the same amount of time.** Please describe whether you think this program will achieve near perfect speedup on a high core count multi-core processor that implements invalidation-based cache coherence. Please describe why or why not? If you think there is a way to improve its speedup substantially, describe how you might improve the code (rough pseudocode is fine).

Solution: Now the cost of the CAS might be high, so the program probably has significant synchronization overhead. We can avoid the write by first checking if `(ball_height < lowest_so_far)` before attempting the CAS. If the check fails, then there is no need to execute the CAS. Another solution might be to compute a local min value and then to update a global min at the end. Note that there were some subtly incorrect answers such as using a global array with an index for each thread. This would not fix the cache-line invalidation issue because of false sharing.

D. (4 pts) Now consider the case where `simulate_random_kick` outputs both the lowest ball height AND a video of the simulated kick for Harry to watch. **Harry implements the following program which is intended to print the lowest kick height as before, and should also saves the corresponding video to disk.** Here's the updated code.

```
int lowest_so_far;   // shared global variable among threads,
                     // holds lowest height so far in centimeters

Video lowest_vid;

void thread_main() {   // assume this code is run by many threads

   float ball_height;
   Video vid;

   for (int i=0; i<1000000/num_threads(); i++) {
     // runs a sim that fills in ball height and corresponding video
     simulate_random_kick(&ball_height, &vid);

     int old_lowest = lowest_so_far;
     int min_height = min(ball_height, old_lowest);

     while (atomicCAS(&lowest_so_far, old_lowest, min_height) != old_lowest) {
       old_lowest = lowest_so_far;
       min_height = min(ball_height, old_lowest);
     }

     if (ball_height == min_height) {
        lowest_vid = vid;
     }
   }

   barrier();     // a regular barrier across all threads

   if (get_thread_id() == 0) {  // assume get_thread_id() behaves as expected
      printf("The lowest height is %d cm\n", lowest_so_far);
      saveFile(lowest_vid);
   }
}
```

Please describe the correctness problem in the code.

Solution: The problem is that the update of `lowest_so_far` and the update of `lowest_vid` is not atomic. As a result, when a CAS succeeds by thread X, there may be an intervening successful CAS and update of `lowest_vid` by another thread prior to thread X being able to write the appropriate value of `vid` to `lowest_vid`. As a result, although `lowest_so_far` will always reflect the lowest ball height found so far, the value of `lowest_vid` may not the be the corresponding video showing the lowest penalty kick.

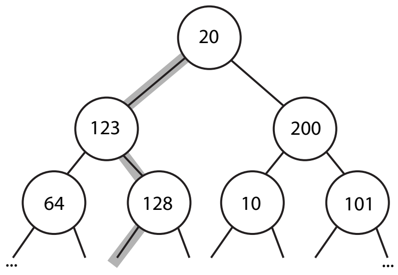
E. (2 pts) Imagine that you are trying to solve the same problem as 3D, but instead of `atomicCAS` you have access to a regular lock via `lock()` and `unlock()`. Please pseudocode a solution that correctly obtains the right answer, while minimizing the amount of time spent in its critical section. Your solution need not rewrite the whole algorithm, just give us the body of the `for` loop.

```
simulate_random_kick(&ball_height, &vid);
lock();
if (ball_height < lowest_so_far) {
  lowest_so_far = ball_height;
  lowest_vid = vid;
}
unlock();
```

Running CUDA Code on a GPU

Problem 4. (14 points):

Consider a complete binary tree of depth 16 that holds one floating point number at each node as shown below. (The figure shows up to depth two.)



Now imagine a CUDA program where each CUDA thread computes the sum of results obtained by applying the functions `f()` or `g()` to all numbers on a path from the root to a leaf. The path taken through the tree is determined by the thread's id, as given in the code below. (In the figure above we highlight the path for thread id 2 which in binary is ...00000010 or left-right-left-left-left...)

```
struct Node {
   Node *left, *right;
   float value;
};

void traverse(Node* root, float* output) {
   int threadId = blockDim.x * blockIdx.x + threadIdx.x;  // compute 1D thread id
   float sum = 0.0;
   int pathBits = threadId;
   int depth = 0;
   Node* curNode = root;
   while (curNode != NULL) {

      // Consider this a single load of 12 bytes
      // Assume processor doesn't use arithmetic cycles to issue loads
      float val = curNode->value;    // 4 bytes
      Node* left = curNode->left;    // 4 bytes
      Node* right = curNode->right;  // 4 bytes

      if (depth % 2 == 0)
        sum += f(val);    // 7 arithmetic instructions
      else
        sum += g(val);    // 7 arithmetic instructions

      // ***** count the lines below as 3 arithmetic instructions
      curNode = (pathBits & 1) ? right : left;  // *** line "A" ***
      pathBits >> 1; // shift right by 1 bit
      depth++;
   }

   output[threadId] = sum;  // each thread writes its result
}
```

Questions are on the next page.

A. (4 pts) Assume that the program runs on a GPU with a SIMD width (aka CUDA warp size) of 32. Does the program suffer from low utilization due to SIMD divergence, why or why not? (For simplicity, please assume that the conditional "?" operator in line A is a single statement with no divergence.)

Solution: There is no divergence. All threads execute the same number of loop iterations, and the value of depth is the same across all threads, so the body of the loop itself has no divergence. Note that even though the threads are operating on different tree nodes, they can share an instruction stream.

B. (5 pts) Now consider the program running on a GPU with one core with a SIMD width of 1 (ignoring SIMD in this problem). Assume that the GPU has memory latency of 100 cycles and infinite memory bandwidth. Also assume the GPU core can one run arithmetic instruction per clock, and supports 8 interleaved CUDA thread execution contexts. **Assuming that the program launches 8 CUDA threads of work, what is the utilization of the GPU when running the `while` loop of this code?** We care only about the steady state behavior of the loop.

Solution: This is a standard interleaved multi-threading scenario where a program issues a memory load with latency 100, and then follows up with 10 arithmetic instructions. Since the processor supports 8-wide multi-threading, when one thread is stalled waiting on memory, the processor has seven other threads that each can contribute 10 cycles of work. As a result, 70 of the 100 stall cycles are hidden, so the utilization of the processor is 70%.

C. (5 pts) Finally, consider the program running on the same GPU as in part B, (1 GHz, 1 core, 1 op/-clock, simd width=1) but now with memory latency = 0, and with 4 GB/sec of bandwidth. What is the utilization of the GPU when running the `while` loop in the code? (You can assume 1 GB is 10^9 bytes.)

Solution: 100% utilization. The arithmetic intensity of the problem is 10 instructions per 12 bytes loaded. 12 bytes per 10 cycles at 1 GHz is 1.2 GB/sec. Since the core has 4 GB/sec of bandwidth, it is compute bound, and runs at 100% utilization.

Implementing Transactions

Problem 5. (16 points):

Below is an implementation of an **optimistic read, pessimistic write, lazy versioning** STM that tracks reads and writes at the granularity of objects.

```
// Assume TMDesc is a transaction descriptor data structure
//    -- use GetDataVersion(obj) to access the version
// Assume each object maintains a lock, which supports these two ops:
//    -- LockObj(obj) returns true if success, false if failure
//    -- CheckLock(obj) returns true if locked, false otherwise
//    -- ReleaseLock(obj)

// helper routines ////////////////////////////////////////////////////

// add object to read set
void OpenForReadTx(TMDesc tx, object obj) {
  tx.readSet.obj = obj;
  tx.readSet.version = GetDataVersion(obj);
  tx.readSet++;
}

// add object to write set
OpenForWriteTx(TMDexc tx, object obj) {
  if(!LockObj(obj) {      // try to lock object for writing
     AbortTx(tx);         // abort if someone else holds the lock
  }
  tx.writeSet.obj = obj;
  tx.writeSet.version = GetDataVersion(obj);
  tx.writeSet++;
}

// offset denotes what field of the object is being written to and needs to be buffered
// aka... conflict detection is at object granularity, but write logging at field granularity.
void writeBuffIntInsertTx(TMDesc tx, object obj, int offset, int value) {
  tx.writeBuff.obj = obj;
  tx.writeBuff.offset = offset;
  tx.writeBuff.value = value;  // buffer the value
  tx.writeBuff++;
}

  // helper: returns the int corresponding to the appropriate offset in object obj
int ReadDataFromMem(object obj, int offset);

// helper: writes value to the appropriate location in memory
void WriteDataToMem(object obj, int offset, int value);

void AbortTx(TMDesc tx); // call this internal helper to abort the transaction

// you will implement ReadIntTx in part A
int ReadIntTx(TMDesc tx, object obj, int offset) { }

// you will implement CommitTx in part A
bool CommitTx(TMDesc tx) { }

// you will implement UnlockObj in part A
void UnlockObj(object obj, TMVersion version) { }
```

A. (12 pts) Provide implementations for `ReadIntTx` (which reads an `int` value), `CommitTx()` (which commits transactions) and `UnlockObj()` (which commits writes) to ensure correct operation of this STM. Pseudocode is fine, but it must be sufficiently precise for the grader. **Hint: Be careful: how do you ensure your reads get the most up to date data?**

```
int ReadIntTx(TMDesc tx, object obj, int offset) {
  // pseudo code to check if matching obj and offset are in write buffer
  if (obj and offset in tx.writeBuff)
    return tx.writeBuff.value;
  else
    return ReadDataFromMem(obj, offset);
}

bool CommitTx(TMDesc tx) {

  // Check read set (on commit because it's optimistic)
  foreach (entry e in tx.readSet) {

    // Check lock to make sure that no other transaction has the lock
    // and therefore has the obj in it's write set , but not
    // yet committed.
    // Also check version to make sure value hasn't changed (due to a commit)
    // by some other transaction.

    if (CheckLock(e.obj) || e.version != GetDataVersion(obj)) {
      AbortTx(tx);
      return false;
    }
  }

  // Flush write-buffer
  foreach (entry e in tx.writeBuff) {
    WriteDataToMem(e.obj, e.offset, e.value));
    tx.writeBuff = 0;  //empty write buffer
  }

  // Unlock write-set
  foreach (entry e in tx.writeSet) {
    UnlockObj(e.obj, e.version);
  }

  return true;
}

void UnlockObj(object obj, TMVersion version) {

  // Increment version
  SetDataVersion(obj, version + 1);

  // Release lock on object
  ReleaseLock(obj);
}

// Note: It's important to flush the ENTIRE write buffer before releasing any locks.
// Otherwise another transaction could write to an object in the current write set
// (that has been unlocked) and commit this result before the current transaction is
// done with it's commit, breaking the atomicity guarantee of transactions.
```

B. (4 pts) **THE REST OF THIS PROBLEM IS INDEPENDENT OF YOUR ANSWERS FROM PART A. NOTICE THAT THE TM SYSTEM IS NOW EAGER.** Assuming **optimistic reads, pessimistic writes and eager versioning**, fill in the tables below for the two concurrent transactions X1 and X2. **Assume all data and version numbers are initialized to zero.** Assume the transactions proceed concurrently, and the operations occur in order listed to the left of each statement. e.g., (1) is the first operation to occur in time, (2) is the second, etc. (5) and (7) are the commit times.

```
     X1                    X2
------------------------------------------------
     atomic {              atomic {
(1)    obj1.x = 5     (2)    t1 = obj1.x
(4)    obj2.x = 6     (3)    t2 = obj2.x
(5)  }                (6)    obj3.x = t2 + 1;
                      (7)  }
```

After step (1) in the figure above, the state of the system looks like this: (The table below shows the metadata for all objects, as well as the read/write sets for all transactions, as well as the state of the transaction undo logs.

	X	version	Locked by
Obj1	5	0	X1
Obj2	0	0	-
Obj3	0	0	-

	Read Set	Write Set	Undo Log
X1	{}	{(obj1, 0)}	{(obj1.x, 0)}
X2	{}	{}	{}

Now please fill in the table to describe the state of the system after step (6) in the figure above:

	X	version	Locked By
Obj1	5	1	-
Obj2	6	1	-
Obj3	1	0	X2

	Read Set	Write Set	Undo Log
X1	{}	{} *or* {(obj1, 0),(obj2, 0)}	{} *or* {(obj1.x,0), (obj2.x, 0)}
X2	{(obj1, 0), (obj2,0)}	{(obj3.x,0)}	{(obj3.x,0)}

Solution: In answering this question (and also in general), students should keep in mind the difference between abstraction and implementation. In the context of this problem you are given a TM system with the following behavior: optimistic reads, pessimistic writes, and eager versionining. ANY implementation of the TM system that correctly implements these behaviors is valid, and there are of course others beyond the specific implementation given in HW5. In this problem we are saying: imagine you are given a particular implementation that ended up executing the instructions in the given order (you should be able to convince yourself that this is possible).

If students answered assuming the instructions were executed in the order of the problem, they get the provided solution.

If students assumed that the underlying implementation was the implementation from the HW, then X1 would abort at (4) and the instructions wouldn't be executed in the order given in the problem. However, if students therefore ignored that order and answered based on the implementation from HW5, they STILL would get the posted answer. If student's explicitly called out a different set of abort behaviors corresponding to a valid implementation of a TM system supporting optimistic reads, pessimistic writes, and eager versionining, we graded on a case-by-case basis, although it was possible to still get full credit.

Breadth-first search again, but this time on specialized hardware

Problem 6. (16 points):

CS149 students loved optimizing BFS in programming assignment 4 and they also love learning about the benefits of specialized hardware. Therefore, the students are *more-than-thrilled* to have the opportunity use what they learned about parallel systems to design specialized hardware for BFS on an exam.

As we discussed in class, modern hardware design involves choosing the right data storage and data communication components for the job. Here are the hardware resources available for your implementation. Keep in mind the DRAM refers to off-chip memory accessed with high latency. SRAM is high performance onchip storage that can be accessed with high bandwidth and low latency.

Storage Datatypes

```
DRAM<T> name(kSize);  // DRAM with element type T, and constant size kSize
SRAM<T> name(kSize);  // SRAM with element type T, and constant size kSize
FIFO<T> name(kSize);  // FIFO with element type T, and constant size kSize
ArgIn<T> name;        // Input scalar of type T
```

FIFO is a queue data structure supporting enqueue (enq) and dequeue (deq) operations (first-in-first out).

```
void fifo.enq(T data);
T fifo.deq();
```

You are also given supporting code for reading from DRAM.

```
// Loads a single int from DRAM (the int at the given offset)
void load(T destination, DRAM<T> source, int offset);

// Loads a contiguous block of sizeof(int) * data from DRAM (source) starting at `offset`
// into SRAM (destination)
void load(SRAM<T> destination, DRAM<T> source, int offset, int numElements);
```

Consider the following unoptimized pseudocode for BFS. **Note to students: although the code looks long, don't worry, you will be able to get your head around it quickly. It is just BFS written in terms of the memory primitives. Recall there are definitions on the previous page.**

```
// Type definitions
typedef int Node;
struct NeighborData  {int degree; int offset;}  // num neighbors + points into adjacency
                                                // list for where to get adjacent node info
struct NodeWithData  {Node node; NeighborData data;} // a graph node + its adjacent info
struct NodeWithLayer {Node node; int layer;}         // a graph node + the "step" BFS finds it

// Constants
int kMaxFrontierSize;  // assumed to be larger than maximum possible frontier size
int numNodes;          // size of graph

DRAM<NodeWithData> nodeDRAM(numNodes); // the graph, stored in DRAM
DRAM<Node> adjacencyDRAM(numEdges);    // adjacent node ids for every node in the graph
ArgIn<NodeWithData> source;            // BFS starting node
FIFO<NodeWithLayer> output;            // results of BFS... for each node, give step it was found.
                                       // it's a fifo because results are streamed from BFS impl

FIFO<NodeWithData> frontier(kMaxFrontierSize);
SRAM<Bit> visited(numNodes);           // bitvector storing whether a node was visited yet

frontier.enq(source);  // Initialize the first frontier
NodeWithLayer initial = {source, 0};
output.enq(initial);   // Push the source node to the output, since it is "found"

for (int layer = 0; ;layer++) { // infinite loop with counter
  int frontierSize = frontier.size;
  if (frontierSize == 0) { break; } // Exit the loop since there's no work left
  for (int i = 0; i < frontierSize; i++) {
    NodeWithData current = frontier.deq();

    // walk neighbor list
    int numNeighbors = current.data.degree;
    for (int neighborNum = 0; neighborNum < numNeighbors; neighborNum++) {

      // get neighbor node id from DRAM
      Node neighborNode;
      load(neighborNode, adjacencyDRAM, current.data.offset + neighborNum);
      if (!visited[neighborNode]) {
        visited[neighborNode] = true; // update bitvector
        NodeWithLayer outputData = {neighborNode, layer + 1};
        output.enq(outputData); // emit to output
        NodeWithData newJob;
        load(newJob, nodeDRAM, neighborNode);
        frontier.enq(newJob);  // place adjacent node on frontier.
      }
    }
  }
}
```

A. (4 pts) Assume that DRAM is organized into DIMMs on a 64-bit memory bus as explained in lecture. (A DIMM consists of 8 DRAM chips, and each chip have transmit 8 bits per clock.) Also recall that DRAM has higher latency when accessing data from a row that is not "activated". What limits the performance of this implementation of BFS? Explain why? **(Hint: take a look at how the implementation accesses data, how is it inefficient? You may even want to contrast this implementation to the optimized implementation in part B.)**

Solution: This implementation spends most of the time waiting for DRAM memory accesses for neighborNode *and* newJob*. Even though DRAM access limits performance, the DRAM is not used efficiently because the solution pays the worst-case latency on every DRAM access and also the implementation does not keep the DRAM busy (it only accesses 4 bytes (32 bits) at a time.*

Now consider the following implementation that is optimized from the one above. The code is largely the same as before, but the `i` loop has been split into an `i` and `innerNeighbor` loop.

```
// Types
typedef int Node;
struct NeighborData  {int degree; int offset;}  // num neighbors + points into adjacency
                                                // list for where to get adjacent node info
struct NodeWithData  {Node node; NeighborData data;} // a graph node + its adjacent info
struct NodeWithLayer {Node node; int layer;}         // a graph node + the "step" BFS finds it

// Constants
int kMaxFrontierSize;  // assumed to be larger than maximum possible frontier size
int numNodes;          // size of graph
int kNeighborChunkSize // "tiling" factor

DRAM<NodeWithData> nodeDRAM(numNodes);  // the graph, stored in DRAM
DRAM<Node> adjacencyDRAM(numEdges);     // adjacent node ids for every node in the graph
ArgIn<NodeWithData> source;             // BFS starting node
OutputFIFO<NodeWithLayer> output;       // results of BFS...

FIFO<NodeWithData> frontier(kMaxFrontierSize);
SRAM<Bit> visited(numNodes);

frontier.enq(source); // Initialize the first frontier
NodeWithLayer initial = {source, 0};
output.enq(initial);  // Push the source node to the output, since it is "found"

for(int layer = 0; ; layer++) { // infinite loop with counter
  int frontierSize = frontier.size;
  if (frontierSize == 0) { break; } // Exit the loop since there's no work left
  for (int i = 0; i < frontierSize; i++) {
    NodeWithData current = frontier.deq();
    int numNeighbors = current.data.degree;

    // process all neighbors, in chunks
    for (int neighborNum = 0; neighborNum < numNeighbors;
                         neighborNum += kNeighborChunkSize) {
      SRAM<Node> neighbors(kNeighborChunkSize);
      int lowerBound = current.data.offset + neighborNum;
      int numToFetch = min(neighborNum + kNeighborChunkSize, numNeighbors);
      int upperBound = current.data.offset + numToFetch;
      load(neighbors, adjacencyDRAM, lowerBound, upperBound);

      // now process the neighbors in the chunk
      for (int innerNeighbor = 0; innerNeighbor < numToFetch; innerNeighbor++) {
        Node neighborNode = neighbors[innerNeighbor];
        bool alreadyVisited = visited[neighborNode];
        if (!alreadyVisited) {
          visited[neighborNode] = true;
          NodeWithLayer outputData {neighborNode, layer + 1};
          output.enq(outputData);
          NodeWithData newJob;
          load(newJob, nodeDRAM, neighborNode);
          frontier.enq(newJob);
        }
      }
    }
  }
}
```

B. (4 pts) Explain why the optimized implementation given here is better than the unoptimized version from part A. (We're looking for two reasons.)

Solution: In the unoptimized implementation, each neighbor is loaded from the DRAM and processed sequentially. This limits the throughput of the unoptimized implementation to the DRAM latency rather than the DRAM bandwidth. To better utilize the DRAM bandwidth, the optimized implementation tiles the neighbor list into chunks, and accesses the DRAM in burst mode a chunk at a time. The worst case DRAM access latency is now paid once per chunk, instead of once per neighbor.

C. (4 pts) The BFS code from part B has three loops. Which of these loops are good candidates for parallelization? What conditions/assumptions do you need to make to parallelize these loops.

Solution:

- *Layer parallelization does not work because loop carried dependencies exist on all data structures*
- *Frontier parallelization results in a data race on the visited set, which can result in extraneous DRAM loads (both bulk and scalar). Since this application is memory-bound, this will negatively affect overall performance. In the case of densely connected graphs, this could cause the implementation to be slower than an unparallelized one. Furthermore, this can also result in duplicated writes to the output stream. A possible solution is to use an atomic test-and-set operation on the visited set to avoid this data race.*
- *Neighbor parallelization: The outer neighbor loop can be parallelized to allow multiple DRAM loads to be in-flight simultaneously. This can help hide DRAM latency, but the benefits depend on the average degree of the graph. The inner neighbor loop can also be parallelized to issue multiple scalar DRAM loads as well. Parallelizing the inner loop also helps hide DRAM latency.*

D. (4 pts) Parallelization will keep the DRAM busy for brief windows of time when the code accessing DRAM is active. To more fully utilize the memory bandwidth we would like to employ pipelining. This allows multiple stages (dependent code blocks) to be active at the same time, and allows compute and memory access to be overlapped in the same loop body. Looking at the optimized code which loops can be pipelined and what benefits does pipelining provide?

Solution:

- *Frontier pipelining is possible, but only overlaps the frontier dequeue with neighbor fetching. Since the dequeue is effectively free, this only provides a marginal reduction in the initiation interval of the loop.*
- *Neighbor (outer loop) pipelining is possible, and overlaps the bulk adjacencyDRAM load with sparse loads from nodeDRAM. This would help saturate memory bandwidth, especially if the two DRAMs are accessed via different channels.*
- *Neighbor (inner loop) pipelining is possible, and would help overlap multiple requests to the node-DRAM. Since this load is a scalar load, overlapping helps significantly with better utilizing memory bandwidth.*
- *Note: Executing multiple layers at a time is possible, but requires rotating between multiple frontier sets, and requires an alternative approach. However, doing so would benefit traversals over deep and narrow graphs.*

.