15-418/618, Spring 2019

Exam 2 Practice SOLUTIONS

April, 2019

Warm Up: Miscellaneous Short Problems

Problem 1. (22 points):

A. (3 pts) A few weeks ago Google released a paper about their Tensor Processing Unit (TPU). This specialized processor is specifically designed for accelerating machine learning computations, in particular, evaluating deep neural networks (DNNs). Give one technical reason why DNN evaluation is a workload that is well suited for fixed-function acceleration. **Caution: be precise about what aspect(s) of the workload are important! Your reason should not equally apply to parallel processing in general.**

Solution: As we discussed in class, DNN evaluation exhibits high arithmetic intensity (a well-written convolution is compute bound on modern CPUs and GPUs), so it would benefit from hardware that provides high-throughput, low-energy arithmetic capability. Since DNN costs are dominated by a small number of operations (kernels like dense convolution), accelerating these specific operations in HW yield efficient performance. (Note: answers such as DNNs require "lots of computation", or access many parameters, correctly state properties of modern deep networks, but didn't identify a reason why an ASIC is a reasonable solution. Bandwidth heavy applications are poor candidates for ASIC acceleration, and parallelism alone doesn't justify a fixed-function solution.

B. (3 pts) Most of the domain-specific framework examples we discussed in class (Halide, Liszt, Spark, etc.) provide **declarative abstractions** for describing key performance-critical operations (processing pixels in an image, iterating over nodes in a graph, etc). Give one performance-related reason why the approach of tasking the application programmer with specifying "what to do", rather than "how to do" it, can be a good idea.

Solution: Declarative abstractions leave implementation details up to the system implementer, so the implementation can be specialized to a particular hardware architecture (whether or not to parallelize, when to use fixed-function hardware, etc.). For example, in class we discussed how a declarative approach to specifying operations on a graph could lend itself to very different implementation on clusters of CPUs vs. a GPU. Similar examples were given for image processing.

C. (3 pts) Consider the implementation of `unlock(int* x)` where the state of the lock is *unlocked* when the lock integer has the value 0, and locked otherwise. You are given two additional functions:

```
void write_fence();  // all writes by the thread prior to this operation are
                     // guaranteed to have committed upon return of this function

void read_fence();   // all reads by the thread prior to this operation are
                     // guaranteed to have committed upon return of this function
```

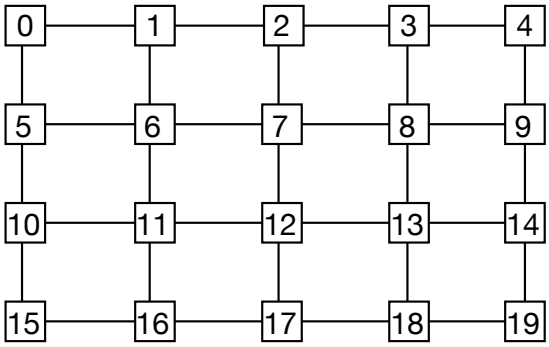
Assume the memory system is a **provides only relaxed memory consistency** where read-after-write (W->R) ordering of memory operations is not guaranteed. (It is not true that writes by thread T must commit before later (independent) reads by that thread.) Provide a correct implementation of `unlock(int* x)` that uses the minimal number of memory fences. **Please also justify why your solution is correct... using the word "commit" in your answer might be a useful idea.**

Solution:

```
void unlock(int* x) {*x = 0;}
```

We require that all writes in the critical section commit (recall that commit means be observable by other processors) prior to the commit of the write to the lock variable to perform the unlock. This ordering is already preserved by the system so no fences are necessary.

Note that in this relaxed memory system, an implementation of `lock(int* x)` would need to ensure that reads after the lock (in the critical section) were not moved up in front of the write that took the lock. Therefore we need to use a read fence in the lock, but this was not part of the question.



D. (3 pts) The Xeon Phi processor uses a mesh network with "YX" message routing. (Messages move vertically in the mesh, undergo at most one "turn", and then move horizontally through the network. Consider two messages being sent on the 20 node mesh shown above. Both messages are sent at the same time. Each link in the network is capable of transmitting one byte of data per clock. Message 1 is sent from node 0 to node 14. Message 2 is sent from node 11 to node 13. **Both messages contain two packets of information and each packet is 4 bytes in size.**

Assume that the system uses store-and-forward routing. You friend looks at message workload and says "Oh shoot, it looks like we're going to need a more complicated routing scheme to avoid contention." Do you agree or disagree? Why?

Solution: Contention does not exist. It will take 3×4=12 cycles before the first packet from Message 1 arrives at node 11. However, after 8 cycles the Message 2 has already completely been transmitted over this link, so the link is free.

E. (4 pts) Now consider the same setup at the previous problem, except the network is modified to use wormhole flow-control with a flit size of 1 byte. (1 flit can be transmitted per link per cycle.) Is there now contention in the network? **Assuming that Message 1 has priority over message 2 in the network, what is the final latency of the end-to-end transmission of Message 2?**

Solution: Three flits of Message 2 leave Node 11 before the first flit of Message 1 arrives. In clock cycle 4 the first flit of message 1 starts transmitting over the 11-12 link. (They have priority over flits from message 2). The link becomes available again 8 cycles later in cycle 12. Cycles 12-16, the final 5 flits of Message 2 leave node 11. With the last flit getting to node 13 in cycle 17. So the total latency is 17 cycles. (A simple solution computes the answer 17 from 8+9. Under no contention, message 2 would have taken 9 cycles to transmit with cut-through flow control. It gets delayed for 8 cycles by message 1.)

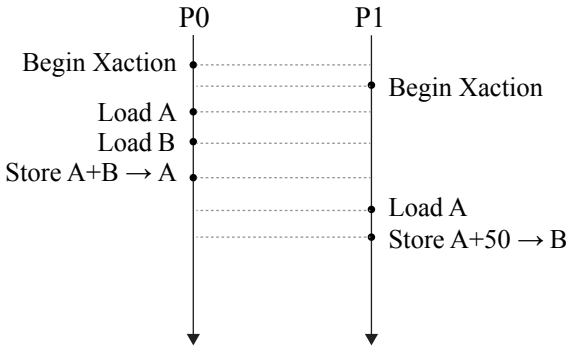
F. (3 pts) You are asked to implement a version of transactional memory that is both **eager and pessimistic**. Given this is a pessimistic system, on each load/store in a transaction T0, the system must check for conflicts with other pending transactions on other cores (let's call them T1, T2). Give a very brief sketch of how the system might go about performing a conflict check for a READ by T0. (A sentence or two about what data structure to check is fine.)

Solution: For each read by T0, the system needs to check for writes to the same address in the undo log of transactions T1 and T2. Any solution that mentioned checking the undo log of other transactions as given full credit.

Transactional Memory

Problem 2. (12 points):

A. Consider the following schedule of operations performed by processors P0 and P1.



Assume that at the start of the program A=0 and B=100 and that this system implements **lazy, optimistic** transactional memory.

Notice that the schedule above does not designate when the end of transactions occur. Fill in the table below for each possible schedule of transaction commits. Indicate whether P0 or P1 (or both) execute a rollback, and fill in the final values of A and B after **both transactions are complete**.

Important! For row 3, you may wish to state your assumptions about the details of the transactional memory implementation to justify your answer.

	P0 rollback (y/n)	P1 rollback (y/n)	A	B
P0 reaches end of transaction before P1 (but after P1's performs loads/stores to A and B)	no	yes	100	150
P1 reaches end of transaction before P0	yes	no	50	50
P0 reaches end of transaction before P1 performs loads/stores to A and B)	no	no	100	150

The situation in row one yields output that is consistent with a serial ordering where P0 computes its work, then P1 observes these results, and then P1 computes its work.

The situation in row two yields output that is consistent with a serial ordering where P1 computes its work, then P0 observes these results, and then P0 does its work.

The provided answer for row three assumes the hardware transactional memory discussed in class. When P0 commits, the read/write set of the pending transaction by P1 does not contain A or B. Thus there is no need to roll back the pending P1 transaction. When this transaction ultimately does read A, it will observe the results of the transaction. Alternatively, an answer that was equivalent to the correct answer to row one was also acceptable, provides a reasonable justification was given.

A Lock-Free Stack

Problem 3. (12 points):

In class we discussed the following implementation of a lock-free stack of integers.

```
// CAS function prototype: update address with new_value if its contents
// match expected_value.  Return value of addr (at start of operation).
Node* compare_and_swap(Node** addr, Node* expected_value, Node* new_value);

struct Node {
  Node* hello;
  int value;
};

struct Stack {
  Node* top;
};

void init(Stack* s) {
  s->top = NULL;
}

void push(Stack* s, Node* n) {
  while (1) {
    Node* old_top = s->top;
    n->next = old_top;
    if (compare_and_swap(&s->top, old_top, n) == old_top)
      return;
  }
}

Node* pop(Stack* s) {
  while (1) {
    Node* top = s->top;
    if (top == NULL)
      return NULL;
    Node* new_top = top->next;
    if (compare_and_swap(&s->top, top, new_top) == top)
      return top;
  }
}
```

A. We talked about how this implementation could fail due to the "ABA problem". What is the ABA problem? Describe a sequence of operations that causes it.

Solution: The ABA problem occurs when the one thread observes the system in state A, then the system changes from state A to B to A', but CAS cannot distinguish between A and A' and incorrectly succeeds. In the stack example, this will be when a node A is initially on the stack and thread 1 attempts to remove A. If prior to thread 1's CAS, another thread removes A, performs other stack operations, and pushes A back on the stack, thread 1's CAS will incorrectly succeed. This will corrupt the stack. The following sequence of operations would corrupt the stack, causing node D to be lost.

Processor 1	Processor 2	The stack
		A->B->
old_top = s->top		A->B->
	A = pop()	B->
	push(D)	D->B->
	push(A)	A->D->B->

```
cas()              |                        |          B->
```

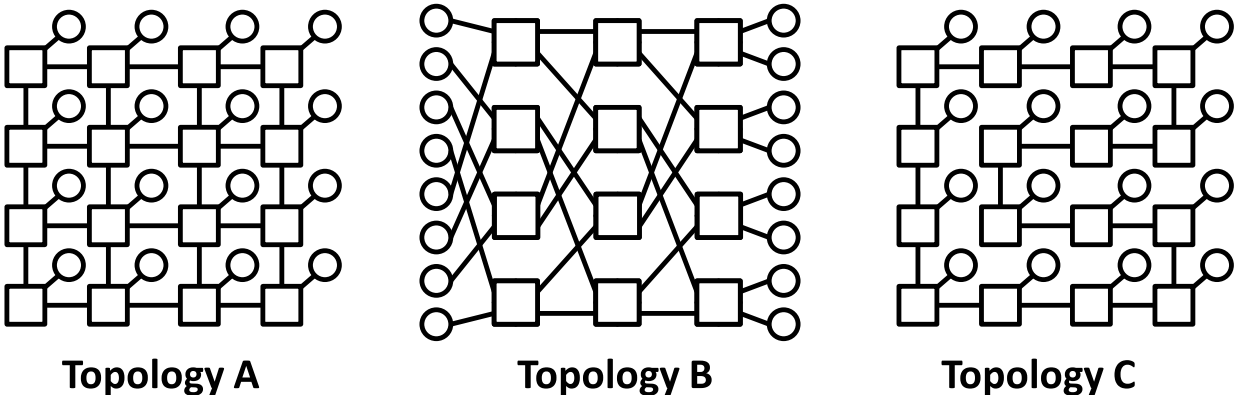
B. Even though the above implementation is lock free, it does not mean it is free of contention. In a system with P processors, imagine a situation where all P processors are contending to pop from the stack. Describe a potential performance problem with the current implementation and describe one potential solution strategy. (A simple descriptive answer is fine.)

Solution: The problem is that "spinning" in the CAS generates a lot of cache coherence traffic. There are a variety of solutions to reduce inefficiencies of spinning. We accepted the solutions "use an exponential back-off strategy" or "perform a non-exclusive read before the CAS" (i.e., a test prior to the compare-and-swap). We also accepted the answer that the problem was a lack of fairness due to the lack of fairness guarantees by the CAS, with the solution being to use a ticket lock system.

Interconnection Networks

Problem 4. (12 points):

A. The figure below shows four common network topologies (circles and squares represent network endpoints and routers respectively).



Identify each topology and fill in the table below. Express the bisection bandwidth in Gbit/s, assuming each link is 1 Gbit/s. Express the cost and latency in terms of the number of network nodes N, using Big O notation, e.g., O(logN)).

	Topology A	**Topology B**	**Topology C**
Topology Type/Name	Mesh	Log. Multi-Stage	Ring
Direct or Indirect	Direct	Indirect	Direct
Blocking or Non-Blocking	Blocking	Blocking	Blocking
Bisection Bandwidth	4 Gbit/s	8 Gbit/s	2 Gbit/s
Cost	O(N)	O(NlogN)	O(N)
Latency	O(sqrt(N))	O(logN)	O(N)

B. Briefly describe two advantages and two disadvantages of circuit-switched networks compared to packet-based networks.

Advantages:
- No need for buffering
- No contention (flow isolation)

Disadvantages:
- Handshake overhead (for setting up and tearing connections)
- Lower link utilization (two flows cannot use the same link)

C. What common networking problem do virtual channels solve?

Solution: Head-of-line blocking

Two Box Blurs are Better Than One

Problem 5. (12 points):

An interesting fact is that repeatedly convolving an image with a box filter (a filter kernel with equal weights, such as the one often used in class) is equivalent to convolving the image with a Gaussian filter. Consider the program below, which runs two iterations of box blur.

```
float input[HEIGHT][WIDTH];
float temp[HEIGHT][WIDTH];
float output[HEIGHT][WIDTH];

float weight;   // assume initialized to (1/FILTER_SIZE)^2

void convolve(float output[HEIGHT][WIDTH], float input[HEIGHT][WIDTH], float weight) {

  for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
      float accum = 0.f;
      for (int jj=0; jj<FILTER_SIZE; jj++) {
        for (int ii=0; ii<FILTER_SIZE; ii++) {

           // ignore out-of-bounds accesses (assume indexing off the end of image is
           // handled by special case boundary code (not shown)

           // count as one math op (one multiply add)
           accum += weight * input[j-FILTER_SIZE/2+jj][i-FILTER_SIZE/2+ii];
        }
      }
      output[j][i] = accum;
    }
  }
}

convolve(temp, input, weight);
convolve(output, temp, weight);
```

A. Assume the code above is run on a processor that can comfortably store `FILTER_SIZE*WIDTH` elements of an image in cache, so that when executing `convolve` each element in the input array is loaded from memory exactly once. What is the arithmetic intensity of the program, in units of math operations per element load?

Solution: It is $FILTER\_SIZE^2/2$*, since each input and output pixel are read exactly once, and each* `convolve` *operation performs* $FILTER\_SIZE^2$ *operations per pixel. We also accepted* $FILTER\_SIZE^2$ *for full credit since the question referred to "per element load".*

It's been emphasized in class the need to increase arithmetic intensity by exploiting producer-consumer locality. But sometimes it is tricky to do so. Consider an implementation that attempts to double arithmetic intensity of the program above by producing 2D chunks of output at a time. Specifically the loop nest would be changed to the following, **which now evaluates BOTH CONVOLUTIONS.**

```
for (int j=0; j<HEIGHT; j+=CHUNK_SIZE) {
  for (int i=0; i<WIDTH; i+=CHUNK_SIZE) {

    float temp[..][..];  // you must compute the size of this allocation in 6B

    // compute required elements of temp here (via convolution on region of input)

    // Note how elements in the range temp[0][0] -- temp[FILTER_SIZE-1][FILTER_SIZE-1] are the tem
    // inputs needed to compute the top-left corner pixel of this chunk

    for (int chunkj=0; chunkj<CHUNK_SIZE; chunkj++) {
      for (int chunki=0; chunki<CHUNK_SIZE; chunki++) {
        int iidx = i + chunki;
        int jidx = j + chunkj;
        float accum = 0.f;
        for (int jj=0; jj<FILTER_SIZE; jj++) {
          for (int ii=0; ii<FILTER_SIZE; ii++) {
            accum += weight * temp[chunkj+jj][chunki+ii];
          }
        }
        output[jidx][iidx] = accum;
      }
    }
  }
}
```

B. Give an expression for the number of elements in the `temp` allocation.

Solution: `(CHUNK_SIZE+FILTER_SIZE-1)`^{2}*. We also accepted* `(CHUNK_SIZE+FILTER_SIZE)`^{2} *for full credit.*

C. Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

Solution: Need $12 \times 12 = 144$ *elements of* `temp` $= 5 \times 5 \times 144 = 3600$ *operations. Producing 64 elements of* `output` *is another* $64 \times 25 = 1600$ *operations. So there are now* $\frac{3600+1600}{64} = \frac{5200}{64} \approx 81$ *operations per pixel, compared to* $2 \times 25 = 50$ *operations per pixel in part A.*

D. Will the transformation given above improve or hurt performance if the original program from part A was *compute bound* for this `FILTER_SIZE`? Why?

Solution: It will hurt performance since it increases the number of arithmetic operations that need to be performed, and the program is already compute bound. Note that a fair number of students said that the problem is was ***arithmetic intensity was increased****, hence the slowdown. Increasing arithmetic intensity of a compute-bound program will not change its runtime if the total amount of work stays the same (it just reduces memory traffic). The essence of the answer here is that more work is being done.*

E. Why might the chunking transformation described above be a useful transformation in a mobile processing setting regardless of whether or not it impacts performance?

Solution: Since the energy cost of data transfer to/from DRAM is significantly higher than the cost of performing an arithmetic operation, reducing the amount of data transfer is likely to reduce the energy cost of running the program. Some students mentioned that reduced memory footprint was also a nice property of the transformed program (it doesn't have to allocate `temp`*), particular since DRAM sizes are smaller on mobile devices. We also accepted this answer for credit.*