15-418/618, Spring 2019

Exam 2 Practice

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.**

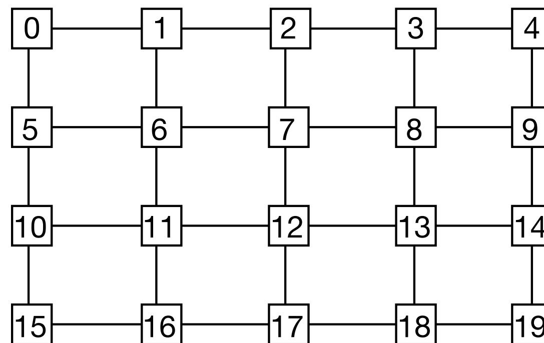
B. (3 pts) Most of the domain-specific framework examples we discussed in class (Halide, Liszt, Spark, etc.) provide **declarative abstractions** for describing key performance-critical operations (processing pixels in an image, iterating over nodes in a graph, etc). Give one performance-related reason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

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.**



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?

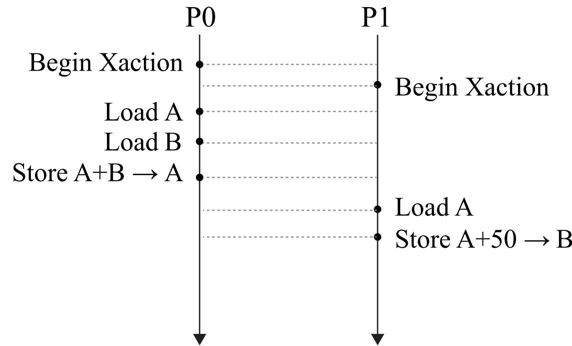
E. (4 pts) Now consider the same setup at the previous problem, except the network is modified to use wormhole flow-control with a flit size of 1 byte. (1 flit can be transmitted per link per cycle.) Is there now contention in the network? **Assuming that Message 1 has priority over message 2 in the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

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.)

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)				
P1 reaches end of transaction before P0				
P0 reaches end of transaction before P1 performs loads/stores to A and B)				

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;
  }
}
```

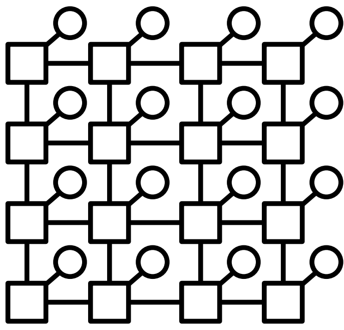
A. We talked about how this implementation could fail due to the “ABA problem”. What is the ABA problem? Describe a sequence of operations that causes it.

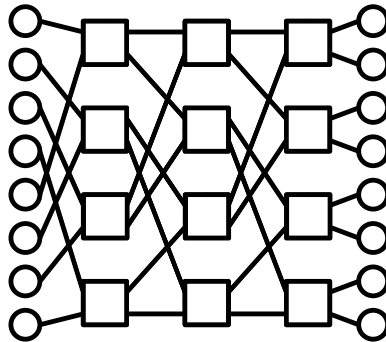
B. Even though the above implementation is lock free, it does not mean it is free of contention. In a system with P processors, imagine a situation where all P processors are contending to pop from the stack. Describe a potential performance problem with the current implementation and describe one potential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

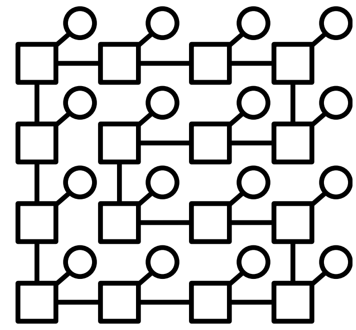
Interconnection Networks

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			
Direct or Indirect			
Blocking or Non-Blocking			
Bisection Bandwidth			
Cost			
Latency			

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

C. What common networking problem do virtual channels solve?

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?

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.

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.

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?

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?