Stanford CS149: Parallel Computing
Written Assignment 5 (THE LAST ONE)
Due Nov 30rd

Problem 1: Implementing Transactions

Below is an implementation of an **optimistic read, pessimistic write, eager versioning** software transactional memory implementation (STM) that tracks reads and writes at the granularity of objects.

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

void OpenForReadTx(TMDesc tx, object obj) {
  tx.readSet.obj = obj;
  tx.readSet.version = GetSTMMetaData(obj);
  tx.readSet++;
}

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

void LogForUndoIntTx(TMDesc tx, object obj, int offset) {
  tx.undoLog.obj = obj;
  tx.undoLog.offset = offset;
  tx.undoLog.value = LowLevel.Read(obj, offset);  // log the value before writing to it
  tx.undoLog++;
}

bool CommitTx(TMDesc tx) {
  // Check read set (on commit because it's optimistic)
  foreach (entry e in tx.readSet) {
    if (!ValidateTx(e.obj, e.version)) {      // you will implement ValidateTx in (a)
      AbortTx(tx);
      return false;
    }
  }

  // Unlock write-set
  foreach (entry e in tx.writeSet)
     UnlockObj(e.obj, e.version);             // you will implement UnlockObj in (a)
  return true;
}
```

A. Provide implementations for `ValidateTx()` (which checks to make sure items in the read set do not conflict with other transactions) and `UnlockObj()` (which commits writes) to ensure correct operation of this STM.

```
// validate obj in read set
bool ValidateTx(object obj, TMVersion version) {









}



void UnlockObj(object obj, TMVersion version) {









}
```

B. Explain how you would need to change the code to implement **pessimistic reads**. (We will accept simple solutions.)

Problem 2: Two Box Blurs are Better Than One

An interesting fact is that repeatedly convolving an image with a box filter (a filter kernel with equal weights, such as the one discussed in class) is equivalent to convolving the image with a "Gaussian filter". (These details are not important, but graphics-inclined students may want to Google.) 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?

Many times in class Prof. Kayvon emphasized 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 temp
    // 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?