1. **(20 points)** Give a one or two sentence definition for each term or phrase below (5 points each).

   (a) False sharing.
   False sharing occurs in a shared-memory multiprocessor if two CPUs access disjoint portions of the same cache line in a way that causes one processor to invalidate (or downgrade from exclusive to shared) the other processors copy and cause a subsequent cache miss (or coherence action).

   (b) Simultaneous multi-threading.
   Simultaneous multi-threading refers to a processor that can execute multiple, independent instruction streams (threads) at the same time. For example, a super-scalar processor can do this by fetching from multiple streams, and using its renaming hardware to keep the instructions for the two streams executing separately.

   (c) One-sided communication.
   This refers to a communication mechanism where one processor can read or write the memory of another processor when the remote processor does not take part in the action. In particular, the remote processor doesn't receive a cache invalidation request, or an interrupt, nor does the remote processor need to respond for the action to finish.

   (d) Tail recursion.
   This refers to a function whose final action is to call another function (typically itself), and the return value from the original function is the value returned by the called function. Tail-recursive calls can be converted into `while`*-loops by a compiler to improve performance and prevent stack-overflows.*

2. **(40 points)** The following questions pertain to the paper, *MapReduce: Simplified Data Processing on Large Clusters*. Each of these questions has a value of 10 points.

   (a) The paper describes a Map-Reduce paradigm. Would it be practical to extend this to Map-Scan? Why or why not?
   The corresponding `scan` *operation would not work in the framework described in the paper.* `Scan` *requires a total ordering of the data values being combined, and the Map-Reduce semantics don't provide such a value. One could imagine adding a second key field,* `k2s` *to the tuples produced by the* `Map` *function, and sorting all of the tuples based on that field. However, this would mean that the* `scan` *part of the job would have to wait until all tuples from the* `map` *phase were computed before it could finish the sort. The extra delay and memory has probably excluded this approach.*

   (b) Briefly describe what happens if a machine taking part in a Map-Reduce computation crashes before the computation is complete. Describe a Map-Reduce computation where the result is the same whether or not some machine crashes. Describe a computation where a different result can be produced because a machine crashes. Your answer to all three of these questions about machine crashes should take *at most* eight sentences.
   If a `map` *node crashes, another node repeats the computation; likewise if a* `reduce` *node computes. If the operations of* `map` *and* `reduce` *are deterministic (including* `reduce` *is independent of the order of its operands) and has no side-effects (e.g. it doesn't write to the global file system), then the final result is unaffected by crashes. Conversely, if* `map` *or* `reduce` *are non-deterministic, check the current time, read and write global files, etc., then the result can be affected by a crash. For example, one could imagine using global files to keep track of the number of* `map` *and* `reduce` *jobs that start, and thereby report the total number of crashes as part of the final answer.*

(c) What would be a reasonable value for λ (from the CTA model) for Map-Reduce computations performed on a large cluster? Given a estimate and justify your answer with one or two sentences.
The paper mentions network delays of a millisecond for communication between machines. The processors are 2GHz Xeons. A Xeon is a superscalar and can (on a good day) execute more than one instruction per cycle, but for simplicity, I'll assume that it executes one instruction per cycle. This is equivalent to one instruction per 0.5ns and yields $\lambda = 1ms/0.5ns = 2$ *million as a reasonable value.*

(d) Here's a common data mining problem. Let $\mathcal{S}$ be a set of sets. For any pair of elements, x, and y, let

$$\text{togetherness}(\mathcal{S}, x, y) \quad = \quad |\{R \in \mathcal{S} \mid (x \in R) \text{ and } (y \in R)\}$$

In English, $\text{togetherness}(\mathcal{S}, x, y)$ is the number of sets in S that contain both x and y. For example, each subset of S could be the set of items purchased in a single transaction at Amazon. In this case, a high value for $\text{togetherness}(\mathcal{S}, x, y)$ would mean that Amazon would tell you that x and y are frequently purchased together.

Write pseudo-code for map and reduce functions that could be used to compute togetherness. You can use the example code from the top of page 108 of the paper as an example for syntax, calling conventions, etc.

```
map(String key, Set value):
   // key:   name of set of values (e.g. identifier for an order)
   // value:   set of items purchased – I'll assume that items
   //    have several fields, and the one called "what" identifies the
   //    kind of thing being purchased. Other fields might give
   //    the price, an inventory code, etc.
   for each p1 in value:
      for each p2 in value:
         EmitIntermediate(new Pair(p1.item, p2.item), "1");

reduce(Pair key, Iterator values):
   // copied from the example from the paper
   int result = 0;
   for each v in values
      result += v;
   Emit(AsString(result));
```

3. **(30 points)** The following questions pertain to the paper, *The GPU Computing Era*. Each of these questions has a value of 10 points.

   (a) Assume that each CUDA core can perform one double-precision, floating-point operation every 4ns. What is the peak-computation rate for the Fermi GPU shown in Figure 3?
   The GPU has 512 CUDA cores. Thus, the peak-computation rate is 128 billion double-precision floating point operations per second.

   (b) A double-precision floating point number is eight bytes. Assume that each double-precision operation has two operands. If each value stored on a Fermi GPU's caches and registers is used once, how long does it take for the GPU to consume all of its on-chip data?
   Each of the 32 streaming-multiprocessors has a 128K byte register file, and 64Kbytes of storage that can be used as L1 cache or shared memory. This gives a total of 32(128K + 64K) = 6.144Mbytes of memory in the streaming multiprocessors. There is an additional 768Kbytes of L2 cache. Tuse the total storage capacity is 6912Kbytes = 864K doubles. With a peak operation rate of 128 billions double-precision operations per second, and each operation consumes two double precision numbers, the Fermi GPU can consume all of its on-chip data in (864K/(2*128billion/second) = 3.456μseconds.*

   (c) A Fermi GPU can read data from off-chip DRAM at a peak rate of 144GB/sec. If a Fermi GPU is to operate at its peak-computation rate, how many double-precision operations must it perform for each double-precision value read from memory? For simplicity, you can pretend that there is nothing to be written back to memory.
   (128G CUDA-ops/second)/((144 G bytes/sec)(1 double/8 bytes) = 7.1 CUDA-ops/double.*

4. **(10 points)** Do any one (but not more) of the following three questions:

   (a) Sketch a way to compute matrix-multiplication using Map-Reduce.
   Let matrices X and Y be stored as blocks. I'll assume there is a standard size for a block, and that the X and Y can have dimensions that are integer multiples of the block dimensions. My computation uses two Map-Reduce passes.
   In the first pass, blocks of Y are read in in the map phase, and each block is output once for each block of X that it must be multiplied by. In particular, if X is $n_1 \times n_2$ (in blocks), and Y is $n_2 \times n_3$, then block (j,k) of Y is output with n_1 times with keys $(1,j)$, $(2,j)$, $\ldots (n_1,j)$. The value field for each of these tuples will be $\{k, Y(j,k)\}$, where $Y(j,k)$ is block (j,k) of Y. In the reduce phase of the first pass, the processor that does the reduce for key (i,j) will compute the product of $X(i,j)$ with $Y(j,k)$ and produce a tuple $\{\{i,k\}, X(i,j) * Y(j,k)\}$. The result of the reduce task for key (i,j) will be k such tuples.
   The second pass will take all the tuples produces by the first pass and use their keys ($\{i,k\}$ above) to sort them for the reduce phase. The reduce phase will compute the sum of its blocks to produce block (i,k) of the result matrix.

   (b) Sketch a way to compute count-3s using CUDA.
   Have each thread compute the sum of the 3's in its portion of the array. The article didn't describe how to do reduce operators in CUDA. So, these values could be returned to the host CPU for the final sum. Or, if you have read more about CUDA, you'll know that CUDA has a built-in reduce operator, and you can use this to compute the final sum. Either approach is acceptable.

   (c) Given an array $x[i]$ for $0 \leq i < n$, use generalized reduce or scan to compute $y[i]$ with

$$y[i] \quad = \quad x[0]^{x[1]^{x[2]^{\cdot^{\cdot^{\cdot^{x[i]}}}}}}$$

   Because $a^{b^c} = a^{bc}$, we get

$$y[i] \quad = \quad x[0]^{\prod_{j=1}^{i} x[j]}$$

   So, we use a scan on the array $x[1 \ldots n-1]$ to compute $\{x[0], \prod_{j=1}^{i} x[j]\}$ for each i in $0 \ldots n-1$. Then, the output function for the scan computes $y[i]$ from these two values.