Name & Student Number:

Final Exam Solution Set

Figure 1 shows a CUDA kernel that we use as an example for questions 1 through 5. These questions are (nearly) independent, so if you get stuck on one that should not stop you from answering the others. The kernel `closest` is launched with two arrays of points, `p` and `q`. For each point in `q`, we want to find the closest point in `p`. The arrays `p` and `q` are stored in global memory before launching the kernel. For each i with $0 \leq i <$ `q_n`, the kernel sets `closest[i]` to the index of the point in `p` that is closest to `q[i]`.

1. **Thread Divergence (8 points)**

   (a) **Thread divergence, definition (2 points)**: What is thread divergence?

   Thread divergence occurs when different threads in the same warp follow different execution paths. For example, one executes the "then-"clause" of an if-statement and the other executes the "else".

   (b) **Thread divergence, pro or con (2 points)**: In general, is it desirable to have thread divergence? Briefly explain why or why not.

   Thread divergence is **undesirable** because the GPU must execute *both* paths, and only some of the pipelines are active for each path. This results in idle exeuction pipelines.

   (c) **Thread divergence, example (4 points)**: Give example of thread divergence from the code in Figure 1. Briefly explain why this is an example of thread divergence.

   The code

   ```
   if(i < p.n) {
     sh_px[i] = p.x[i];
     sh_py[i] = p.y[i];
     sh_pz[i] = p.z[i];
   }
   ```

   Generally, when there are more threads than elements of `p`, the assignments to `sh_px` will only be execute by some of the threads. If `p.n` is not a multiple of the warpsize (i.e. 32), then this will cause thread divergence.
   Other examples of thread divergence include the similar case for `q`:

   ```
   if(i < q.n) { ... }
   ```

   and the check for updating the minimum distance:

   ```
   if((d2 < d2_min) || (j_min == -1)) { // found a closer point
     ...
   }
   ```

   The for-loop,

   ```
   for(int j = 0; j < p.n; j++) { ... }
   ```

   is not an example of thread divergence because `p.n` is the same for all threads.

2. **Global memory references (8 points)**

(a) **Coalesced global memory references, definition (2 points)**: What does it mean for memory references to be coalesced?

If the threads of a warp access *consecutive* locations of the global memory, the memory reference is said to be *coalesced*. Note: I copied that from the solution set for the 2016W2 final. You could do the same – the test was open book.

(b) **Coalesced global memory references, pro or con (2 points)**: In general, is it desirable to have coalesced memory references? Briefly explain why or why not.

Coalesced reference **are** desirable. The GPU can take advantage of accessing a large, contiguous block of memory and achieve high bandwidth with the data transfer. Conversely, if locations are not coalesced, then several bank accesses may be needed, and several transfers of data from the DRAM to the GPU. Thus, coalesced references are handled faster than non-coalesced ones. Note: also from the 2016W2 solution set.

(c) Consider the statement: `sh_px[i] = p.x[i];` in the code from Figure 1.

- (1 points) Circle the global memory access in this statement.

  `sh_px[i] =`$\boxed{\texttt{p.x[i]}}$`;`

- (1 points) Is this a coalesced access (circle one)? [✓ YES] NO
- (2 points) Briefly justify your answer to whether or not the access is coalesced.

  The threads in a warp have consecutive values of `i` because they differ by `threadIdx.x`. This means that they access 32 consecutive floats from the global memory.

3. **Shared memory references (8 points)**

(a) **Bank conflicts, definition (2 points)**: What is a bank conflict when accessing shared memory?

The shared memory of an SM on a (nVidia) GPU is partioned into *banks* to allow multiple, parallel accesses. A bank conflict occurs when two or more threads in a warp attempt to access different locations of the same bank with the same instruction.

(b) **Bank conflicts, pro or con (2 points)**: In general, is it desirable to have bank conflicts? Briefly explain why or why not.

Bank conflicts **are not** desirable. The multiple references to the same bank must be executed sequentially, making a load or store take up to warpsize (32) clock cycles instead of 1.

(c) Consider the statement `float dx = qx - sh_px[j];` in the code from Figure 1.

- (1 points) Circle the shared memory access in this statement.

  `float dx = qx -`$\boxed{\texttt{sh\_px[j]}}$`;`

- (1 points) Does this access have a bank conflict (circle one)? YES [✓ NO]
- (2 points) Briefly justify your answer to whether or not the access has a bank conflict.

  All threads in a warp execute the `for(int j...)` loop in lockstep. Thus, they all have the same value of `j` and all access the same location of the same bank of the shared memory. This is not a conflict.

4. **CGMA (12 points)**

(a) **CGMA, definition (1 point)**: What does CGMA stand for?

Compute to Global Memory Access (ratio)

(b) **Find the global memory accesses (3 points)**: Identify the global memory accesses in the `closest` kernel from Figure 1. You can just copy each statement with a global access, and circle the part of the statement that is the global access. Include both reads (i.e. loads) and writes (i.e. stores).

```
sh_px[i] = p.x[i];
sh_py[i] = p.y[i];
sh_pz[i] = p.z[i];
float qx = q.x[i];
float qy = q.y[i];
float qz = q.z[i];
closest[i] = j_min;
```

.

(c) **How many global memory accesses (1 point)**: What is the maximum number of global memory accesses performed by one thread of the `closest` kernel? We are asking for the maximum, so you do not need special cases depending on the control flow when executing the thread. Your answer may depend on `MAX_N_P`, `p.n` and/or `q.n`.

> Each of the global memory references listed above is performed *once* when the thread executes. Thus, there are 7 references. Note: in the version of the `closest` kernel handed out at the exam, the write to `closest[i]` was inadvertently in the body of the `for` loop. This was caught within the first few minutes of the exam, a correction was put on the projector, and an announcement was made. If a student treated this assignment as if it were in the body of the `for` loop, then the thread would make `p.n + 6` global memory references. This answer will be given full credit if the derivation is clearly stated.

(d) **Find the floating point operations (3 points)**: Identify the floating point operations in the `closest` kernel from Figure 1. You can just copy each statement with a floating point operation, and circle the part of the statement that is the floating point operation. Include add, subtract, multiply, divide, or comparison.

```
float dx = qx ⊖ sh_px[j];
float dy = qy ⊖ sh_py[j];
float dz = qz ⊖ sh_pz[j];
float d2 = dx⊛dx ⊕ dy⊛dy ⊕ dz⊛dz;
if((d2 ⧀ d2_min) || (j_min == -1)) { ...  }
```

(e) **How many floating point operations (1 point)**: What is the maximum number of floating point operations performed by one thread of the `closest` kernel? We are asking for the maximum, so you don't need special cases depending on the control flow when executing the thread. Count adds, subtracts, multiplies, divides, and comparisons as one operation each. Count any fused multiply adds as two operations. Your answer may depend on `MAX_N_P`, `p.n` and/or `q.n`.

> Each of the nine floating point operations indicated above is executed once for each element of `p` (assuming that `i < q.n`). Thus, $9 \times$ `p.n` floating point operations are performed by the thread.

(f) **CGMA (1 point)**: What is the CGMA for this kernel? You should calculate this based on the maximums you determined for global memory accesses and floating point operations above. Your answer may depend on `MAX_N_P`, `p.n` and/or `q.n`.

$\frac{9}{7} \times \texttt{p.n}$

(g) **Getting a good CGMA (2 points)** Assume that we need a CGMA of 60 to get good performance. Is it possible for the `closest` kernel to achieve this CGMA? Briefly justify your answer.

Yes. We just need to have enough elements in `p`.

$$\begin{array}{rl} & \frac{9}{7}(\texttt{p.n}) \geq 60 \\ \Rightarrow & \texttt{p.n} \geq \frac{7}{9}(60) \quad = \quad 46\frac{2}{3} \end{array}$$

5. **CUDA and data structures (10 points)**

The code in Figure 1 uses `struct Points` that has fields that are pointers to arrays of floats, i.e. `x`, `y`, and `z`. This is a "struct of arrays" organization of the data. Often, C programmers will write an "array of structs" organization; for example:

```
struct Point {
  float x, y, z;
};
```

With this representation we can write a new version of the kernel where the shared memory array for `p` becomes:

```
__shared__ float sh_p[MAX_N_P];
```

Changing the kernel `closest` to take arrays of `Point`s and their sizes as arguments, we get:

```
__global__void closest(struct Point *p, int p_n,
                       struct Point *q, int q_n, int *closest) {
          ...
};
```

The statements to load `p` become

```
if(i < p_n) {
  sh_p[i].x = p[i].x;
  sh_p[i].y = p[i].y;
  sh_p[i].z = p[i].z;
}
```

The assignments to the three fields are written separately because that is what the compiler will do and it makes the questions below easier.

(a) **Coalescing global memory references (4 points)**: Consider the statement:

```
sh_p[i].x = p[i].x;
```

In the new version of the kernel, is the global memory reference in this line coalesced? Briefly justify your answer based on how an array of structs is stored in memory.

The global memory references are **not** coalesced. Each `Point` struct consists of three `float`s. Thus, successive elements of `p[i].x` are separated by `3*sizeof(float)`. To access the 32 values of `p[i].x` for a warp, the GPU will need to read $32 * 3 *$ `sizeof(float)` $= 32 * 3 * 4 = 384$ bytes, only $\frac{1}{3}$ of which is used.

(b) **Shared memory bank conflicts (4 points)**: Consider the statement:

```
float dx = qx - sh_p[j].x;
```

In the new version of the kernel, does this access have a bank conflict? Briefly justify your answer.

No. As in the previous answer, all threads in a warp will access the same location when loading `sh_p[j].x`. There is no conflict.

(c) **Observations (2 points)**: Based on your answers above, which approach do you expect to have **better** performance? Briefly justify your answer.

Better performance (circle one): [✓ "struct of arrays"] "array of structs"

Although both versions are free of shared memory bank conflicts, the "struct of arrays" version makes significantly better use of the global memory bandwidth because the global reads are fully coalesced.

6. **Erlang Reduce (14 points)**: Figure 2 provides code implementing a reduce problem in CUDA. In this question you will translate it into Erlang. The Erlang version `cluster_mean_erl()` is given in figure 3, but you need to provide the helper functions. In answering the questions below, you may find it useful to refer to figure 5, which provides the complete Erlang code for a similar reduction.

In answering this question, it was useful to observe that the CUDA code broke down nicely into three chunks corresponding to the `leaf()`, `combine()` and `root()` functions in Erlang. The summary information needed for each sublist is the `sum` and `count` of the matches in that sublist.

The first loop (over `k`) collects this information for those elements of the entire list assigned to this thread, which is the job of `leaf()` in the Erlang version. (Note that those elements are *not* consecutive: They are chosen with stride `b` so that the global memory accesses within the loop will be coalesced across the threads on each iteration. But that is a CUDA optimization thing with no relevance to the Erlang implementation).

The second loop (over `stride`) implements the reduction tree to pull together all the local summary data, which is the job of `combine()` in the Erlang version. (This code is almost verbatim from the K&H implementation of reduce in figure 5.16.)

The final three lines implement the calculation of the final result based on the summary information from the entire list, which is the job of `root()` in the Erlang version. (Note that `mean` is an array of size 1, but since only thread 0 is writing that is not a problem.

(a) (6 points) Write `cm_leaf(Qx, Closest, Cluster)` where

- `Qx` is the subset of the list of floats corresponding to the array `qx` in `cluster_mean_cuda()` that was assigned to this worker node.
- `Closest` is the same subset of the list of non-negative integers corresponding to the array `closest` in `cluster_mean_cuda()`; for example, if `Qx` contains elements 42 to 137 of `qx`, then `Closest` will contain elements 42 to 137 of `closest` in the same order.

- `Cluster` is a non-negative integer.

Note that we do not need to pass `n` as an argument since the lists encode their own length. The return value will be your summary of the subset of data assigned to this worker node.

We need to implement the first loop from `cluster_mean_cuda()`. Here is a very short version using a list comprehension:

```
cm_leaf(Qx, Closest, Cluster) ->
  Match = [X || {X, I} <- lists:zip(Qx, Closest), I == Cluster],
  {length(Match), lists:sum(Match)}.
```

Here is a longer version using basic pattern matching and (tail) recursion:

```
cm_leaf(Qx, Closest, Cluster) ->
  cm_leaf(Qx, Closest, Cluster, 0,0.0).
cm_leaf([], [], _Cluster, Summary) -> Summary;
cm_leaf([HQx|TQx], [Cluster|TClosest], Cluster, {Count,Sum}) ->
  cm_leaf(TQx, TClosest, Cluster, {Count+1,Sum+HQx});
cm_leaf([_HQx|TQx], [_HClosest|TClosest], Cluster, Summary) ->
  cm_leaf(TQx, TClosest, Cluster, Summary).
```

There are several other possible implementations.

(b) (3 points) Write `cm_combine(Left, Right)`, where `Left`, `Right` and the return value will be subset summary information.

We need to implement the body of the second loop from `cluster_mean_cuda()`.

```
cm_combine({LeftCount, LeftSum}, {RightCount, RightSum}) ->
  {LeftCount+RightCount, LeftSum+RightSum}.
```

(c) (1 points) Write `cm_root(RootSummary)`, where `RootSummary` will be the summary information for the entire data set and the return value should be the desired mean. You may assume that at least one element of `Closest` matched `Cluster`.

We need to implement the final line from `cluster_mean_cuda()`. The assumption ensures that we do not encounter a divide by zero error.

```
cm_root({RootCount, RootSum}) -> RootSum/RootCount.
```

(d) (2 points) Briefly explain what could go wrong if we omitted the `__syncthreads();` command inside the second loop in the CUDA implementation.

During the first loop, each thread writes its local sum and count into the shared memory. During the second loop, a subset of threads combine the sums and counts of other threads in the shared memory. Critically, during the second loop *other* threads will read these sum and count values. Without the call to `__syncthreads()`, a thread might read a value from the shared memory before its final value has been written by a different thread.

(e) (2 points) Briefly explain why we do not need some version of `__syncthreads();` in our Erlang implementation.

In Erlang, synchronization is implicit with messages. The reader of data for a `cm_combine` operation waits to receive messages from its subtrees. These messages are only sent when the data is available.

7. **Sorting Networks (8 points)**

(a) (**2 points**) Consider the input:

`in[0] = 14; in[1] = 3; in[2] = 12; in[3] = 18; in[4] = 5;`

Does the sorting network shown in Figure 4 sort this input correctly? Correctly means that `out[0..4]` is a permuation of `in[0..4]`, and that

`out[0]` $\leq$ `out[1]` $\leq$ `out[2]` $\leq$ `out[3]` $\leq$ `out[4]`

If not, specify at least one pair `out[i]` and `out[j]` (`i != j`) which are incorrectly ordered.

For this input, the output values are:

`out[0] = 3; out[1] = 12; out[2] = 5; out[3] = 14; out[4] = 18;`

See Figure 6. These inputs **are not** sorted correctly. For example, `out[1] = 14 > out[2] = 5`.

(b) (**2 points**) Consider the input:

`in[0] = 5; in[1] = 18; in[2] = 3; in[3] = 12; in[4] = 14;`

Does the sorting network shown in Figure 4 sort this input correctly? If not, specify at least one pair `out[i]` and `out[j]` (`i != j`) which are incorrectly ordered.

For this input, the output values are:

`out[0] = 3; out[1] = 5; out[2] = 12; out[3] = 14; out[4] = 18;`

See Figure 7. These inputs **are** sorted correctly.

(c) (4 points) Give an input consisting only of `0`s and `1`s that the sorting network shown in Figure 4 does not sort correctly.

We consider the example from question 7a and note that we can use any value between 5 and 14 as a threshold for the "Monotonicity Lemma". For example if we map all values that are less than 10 to zero and all values that are greater than or equal to 10 to one, then the input from question 7a becomes:

`in[0] = 1; in[1] = 0; in[2] = 1; in[3] = 1; in[4] = 0;`

If this input is applied to the sorting network from Figure 4 the output is:

`out[0] = 0; out[1] = 1; out[2] = 0; out[3] = 1; out[4] = 1;`

which is not sorted correctly.

8. Short answer questions (18 points).

(a) (3 points) In one sentence, describe the key property of a problem which makes it "embarrassingly parallel." Give **one** type of overhead which is **not likely** to have a major impact on embarrassingly parallel problems, and **one** type of overhead which is **likely** to impact at least some embarrassingly parallel problems. Answers which give more than one of each will receive zero.

An embarrassingly parallel problem is one that can be broken into a large number of subtasks that require very little communication or synchronization to solve the problem.

Overheads accepted as *not likely*:

- Communication.
- Synchronization.
- Resource contention.

Overheads accepted as *likely*:
- Idle processes.
- Extra computation.
- Extra memory.

(b) (2 points) Give **one** reason why Amdahl's speedup formula is generally too **optimistic** (in other words, it predicts **greater** speedup than is actually possible) when the number of processors increases. Answers which give more than one reason will receive zero.

Amdahl's law assumes that the part of the code that can improve from parallel computation achieves *perfect* speed-up; in other words, that portion of computation achieves a speed-up of P when executed with P processors. In reality, parallel overheads (such as those mentioned above) keep us from achieving such a speed-up.

(c) (4 points) In question 6 you computed the mean value of a subset of a data array: The mean of all `qx` values whose corresponding `closest` indexes were equal to the value `cluster`. In real applications, we want to compute a separate mean for every distinct index that appears in `closest`. If the input data were stored as `(Key1, Value1) = (closest[i], qx[i])` pairs for all `i`, would this problem of computing a mean for all possible values of `closest[i]` be suitable for implementation in a map-reduce framework? If so, briefly explain what `(Key2, Value2)` pairs you would use. If not, briefly explain why not.

This problem is a good candidate for map-reduce.
A simple but inefficient solution: Use `Key2 = Key1` and `Value2 = Value1`. The map workers will send all of the points for each key to the reduce worker responsible for that key. The reduce worker will then compute the average of the values for its cluster of points.
A slightly more efficient solution: Have the map workers collect all `Value1` that have common `Key1`. Then `Key2 = Key1` but `Value2 =` list of (`Value1` corresponding to `Key1`). That way each map worker produces only one pair for each `Key1`, and many fewer pairs need to be shuffled and reduced at the next stages.
A significantly more efficient solution: Have the map workers perform a sum reduce over all pairs that have common `Key1`. Then `Key2 = Key1` but `Value2 =` sum and count of (`Value1` corresponding to `Key1`). That way each map worker produces only one pair for each `Key1` and that pair contains only three numbers, Many fewer pairs need to be shuffled and reduced at the next stages, and each pair that is shuffled and reduced is potentially much smaller.

(d) (3 points) In the first half of the term we tested message passing Erlang code on `bowen` and `thetis`, which are a shared memory multiprocessors. In fact, it is quite common to use software written for message passing on shared memory hardware. In a few sentences, briefly explain why the opposite is **not** true: Executing shared memory software on message passing hardware is much less common than the other way around.

Communication cost ($\lambda > 1$). Shared memory computations implicitly move data between processors when it is referenced. This is typically implemented using some kind of cache coherence protocol. The latencies to transfer data between processors in a message passing machine tend to be much larger. The programmer needs to avoid long waits for data. Thus, message passing machines tend to be programmed using a message passing protocol where the programmer can control when data is sent and received.

(e) (2 points) We saw in class that the CUDA GPUs were capable of impressive speedups on matrix-matrix multiplication in which $\Theta(N^{3/2})$ floating point operations were performed on $\Theta(N)$ data. Briefly explain why many algorithms which require $\mathcal{O}(N)$ floating point operations on $\Theta(N)$ data do not achieve large speedups on GPUs.

Memory bandwidth tends to be a severe performance limiter for GPU computations. With an $\mathcal{O}(N)$ algorithm, only a fixed (and usually small) number of operations are performed on each data value. The time to move the data from the CPU to the GPU's global memory, and for the GPU chip to fetch the data from its global memory dominates the time to perform the computation part of the algorithm. (The exception would be any $\mathcal{O}(N)$ algorithm which performs a constant but very large number of operations on each data value, such as the recurrence relation we studied in homework 5.)

(f) (4 points) In 1–2 sentences briefly explain what is a "fused multiply add". Then give an example of a fused multiply add from the code in figure 1: Copy the line of code into your answer below and then circle the operations which are performed as a fused multiply add.

You can take at least two viewpoints to answering this question.

- Hardware: A fused multiply-add (FMA) is where the floating point hardware can perform a floating point multiplication followed by an addition (e.g. `a*x + b`) in the same time as just doing the multiply.
- Software: From a programmer's perspective, if code is written to use FMA, a computer can achieve up to twice the performance of what it would if these operations need to be performed separately.

For grading, we will give full credit for the hardware answer or the software answer; you did not have to provide both.

In the line

```
float d2 = dx*dx + dy*dy + dz*dz;
```

I have circled the two multiply-adds that can be fused.

Although we have mostly talked about floating point FMA, CUDA GPUs also provide an integer FMA instruction. An example of that is the entire line

```
int i = blockDim.x*blockIdx.x + threadIdx.x; // my index
```

.

Do not write your answers on this page—it will not be graded. You may find it convenient to tear this page off when answering exam questions. If you tear it off, do not submit it with the rest of your exam.

```
struct Points {
  int n;
  float *x, *y, *z;
};

#define MAX_N_P 512
__shared__ float sh_px[MAX_N_P];
__shared__ float sh_py[MAX_N_P];
__shared__ float sh_pz[MAX_N_P];

// assume p.n < q.n and p.n <= MAX_N_P
// p.x, p.y, p.z, q.x, q.y, q.z and closest are arrays in global memory
__global__ void closest(struct Points p, struct Points q, int *closest) {
  int i = blockDim.x*blockIdx.x + threadIdx.x;  // my index
  // copy p from global memory to shared memory
  if(i < p.n) {
    sh_px[i] = p.x[i];
    sh_py[i] = p.y[i];
    sh_pz[i] = p.z[i];
  }
  __syncthreads();
  // find closest point in p to our point, (q.x[i], q.y[i], q.z[i]).
  if(i < q.n) {
    float qx = q.x[i];
    float qy = q.y[i];
    float qz = q.z[i];
    int j_min = -1;      // index of closest point in p to our point (so far)
    float d2_min = 0.0f; // square of distance to p[j_min]
    for(int j = 0; j < p.n; j++) {
      float dx = qx - sh_px[j];
      float dy = qy - sh_py[j];
      float dz = qz - sh_pz[j];
      float d2 = dx*dx + dy*dy + dz*dz;
      if((d2 < d2_min) || (j_min == -1)) { // found a closer point
        d2_min = d2;
        j_min = j;
      }
    }
    closest[i] = j_min; // record the index of the closest point in p
  }
}
```

Figure 1: Code for finding the closest point in `p` for each point in `q`. This code is used in questions 1 to 5.

Do not write your answer on this page—it will not be graded. You may find it convenient to tear this page off when answering the questions. If you tear it off, you need not submit it with the rest of your exam.

```
__global__ void cluster_mean_cuda(float *qx, uint *closest, uint n,
                                  uint cluster, float *mean) {

  __shared__ float sum[SIZE];
  __shared__ uint count[SIZE];

  uint i = threadIdx.x;
  uint b = blockDim.x;
  uint num_elem = ceil((double)n / (double)b);

  sum[i] = 0.0;
  count[i] = 0;

  for(uint k = 0; k < num_elem; k++) {
    uint elem = k * blockDim.x + i;
    if(elem < n)
      if(closest[elem] == cluster) {
        sum[i] += qx[elem];
        count[i]++;
      }
  }

  for(uint stride = b / 2; stride >= 1; stride = stride / 2) {
    __syncthreads();
    if(i < stride) {
      sum[i] += sum[i + stride];
      count[i] += count[i + stride];
    }
  }

  __syncthreads();
  if(i == 0)
    mean[i] = sum[i] / count[i];
}
```

Figure 2: CUDA code for the reduction problem for question 6.

Do not write your answer on this page—it will not be graded. You may find it convenient to tear this page off when answering the questions. If you tear it off, you need not submit it with the rest of your exam.

```
cluster_mean_erl(W, Cluster) ->
    wtree:reduce(W,
                 fun(ProcState) ->
                         cm_leaf(wtree:get(ProcState, qx_atom),
                                 wtree:get(ProcState, closest_atom),
                                 Cluster) end,
                 fun(Left, Right) -> cm_combine(Left, Right) end,
                 fun(RootSummary) -> cm_root(RootSummary) end
                ).
```

You may assume that

- `W` is a list of worker nodes arranged in a tree.
- Subsets of the input data corresponding to `qx` and `closest` in `cluster_mean_cuda()` are stored as lists in the `ProcState` of the workers under the keys `qx_atom` and `closest_atom`. Note that different workers may have subsets of different sizes.
- `Cluster` is a non-negative integer.

Figure 3: The Erlang version of `cluster_mean_cuda()` for question 6.

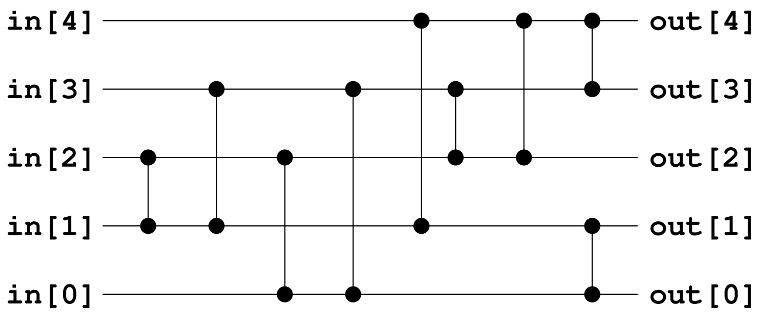

Figure 4: The sorting network for question 7. WARNING: it does not sort correctly!

Do not write your answer on this page—it will not be graded. You may find it convenient to tear this page off when answering the questions. If you tear it off, you need not submit it with the rest of your exam.

```
-module(count_number).

-export([count_number/2, cn_leaf/2, cn_leaf/3, cn_combine/2, cn_root/1]).

count_number(W, Number) ->
    wtree:reduce(W,
                 fun(ProcState) ->
                         cn_leaf(wtree:get(ProcState, data_atom),
                                 Number) end,
                 fun(Left, Right) -> cn_combine(Left, Right) end,
                 fun(RootSummary) -> cn_root(RootSummary) end
                ).

cn_leaf(Data, Number) -> cn_leaf(Data, Number, 0).

cn_leaf([], _Number, Count) -> Count;
cn_leaf([ Number | T ], Number, Count) -> cn_leaf(T, Number, Count + 1);
cn_leaf([ _H | T ], Number, Count) -> cn_leaf(T, Number, Count).

cn_combine(LCount, RCount) -> LCount + RCount.

cn_root(Count) -> Count.
```

Figure 5: Erlang code for implementing a reduction which counts the number of times the value `Number` appears in all the sublists stored under the key `data_atom` in the `ProcState` of the workers in `W`. For example, `count_number(W, 3)` would count the number of 3s in the sublists. You may find this code helpful as a template when answering question 6.

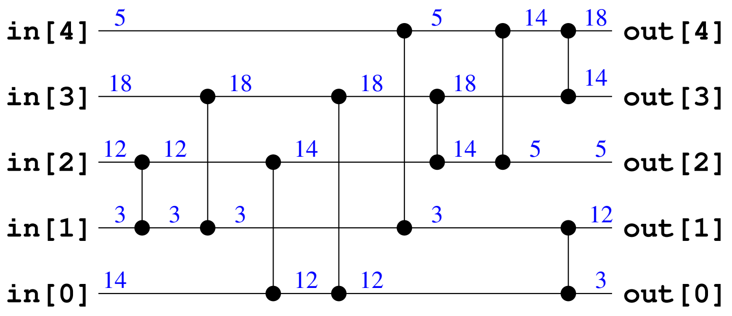

Figure 6: The sorting network from Figure 4 with the input from question 7a

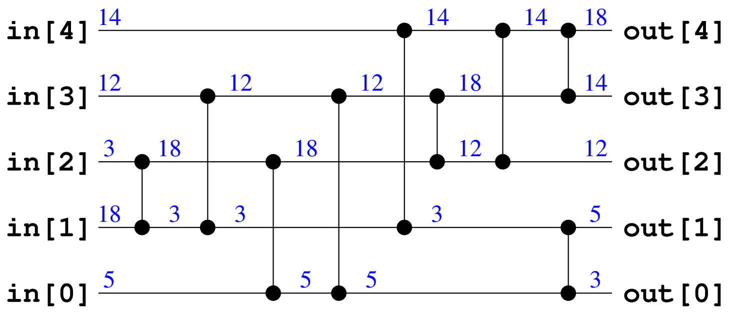

Figure 7: The sorting network from Figure 4 with the input from question 7c