Name & Student Number:

CPSC 418: Parallel Computing 2017 Term 1

Final Exam

Further exam instructions:

- You have 150 minutes (2.5 hours) to complete the exam. There are 86 points available.
- Answer all of the questions in the exam booklet. Keep your answers brief—they should fit in the space provided.
- Figures are at the end of the exam so you can rip them off and have them next to the questions as you write your answers. If you rip off these pages, you do not need to hand them in. **Do not write any answers on the figure pages—they will not be graded.**
- 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?

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

   (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.

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

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

   (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.

   (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.
   - (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.

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

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

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

   (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.
   - (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.

4. **CGMA (12 points)**

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

   (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).

   (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`.

   (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.

(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`.

(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`.

(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.

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__ struct Point 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.

(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.

(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"

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.

   (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.

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

   (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`.

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

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

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.

   (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.

   (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.

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.

   (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.

(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.

(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.

(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 $\Omega(N)$ floating point operations on $\Theta(N)$ data do not achieve large speedups on GPUs.

(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.

This page is an extra in case you run out of space when answering the exam questions. If you use this page, make sure you specify which question(s) you are answering here.

This page is an extra in case you run out of space when answering the exam questions. If you use this page, make sure you specify which question(s) you are answering here.

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.

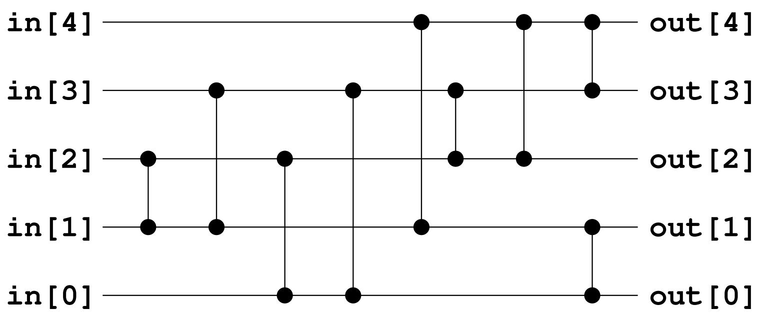

Figure 4: The sorting network for question 7. WARNING: it does not sort correctly!

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.