CPSC 418: Parallel Computing 2017 Term 2

Final Exam

Further exam instructions:

- You have 150 minutes (2.5 hours) to complete the exam. There are 80 points available.
- Answer all of the questions on the exam pages. 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.**

1. (19 points) **CUDA Reduce** Figure 1 provides code implementing a reduce problem in CUDA: Store into vector `y` the `m` largest values in the vector `x` (whose length is `n`). For your convenience, a pure CPU version has also been supplied in figure 2. In answering the questions below, you may assume:
   - The function `reduce_kernel()` is launched with `gridDim = (1, 1, 1)` and `blockDim = (block_size, 1, 1)`.
   - The value of `block_size` is a power of 2 and $32 \leq$ `block_size` ≤ 1024.
   - The input array `x` contains `n` positive floating point numbers and the output array `y` has been allocated of size `m` but has not been initialized.
   - The values of `n`, `m`, `block_size` and the constant `SIZE` have been chosen so that there are no array indexing or out of memory errors.

   For each of the questions below, briefly justify your answer in a sentence. Answers which do not include a brief justification may not receive full marks.

   (a) (2 points) Consider the line `shared[j*b + i] = 0.0;` in section 1 of `reduce_kernel()`. Will this memory access cause a bank conflict? Briefly explain why or why not.

   (b) (2 points) Consider the line `float curr = x[elem];` in section 2 of `reduce_kernel()`. Will this memory access cause a bank conflict? Briefly explain why or why not.

(c) (2 points) Consider the line `float curr = x[elem];` in section 2 of `reduce_kernel()`. Will this memory access be coalesced? Briefly explain why or why not.

(d) (2 points) Consider section 2 of `reduce_kernel()`. Give **one** reason why it is a good idea to use shared memory to store the array `shared[]` for this section of the code. Answers which give more than one reason will receive zero.
Would this code still work if `shared[]` were in global memory?

(e) (2 points) Consider section 3 of `reduce_kernel()`. Give **one different** reason why it is a good idea to use shared memory to store the array `shared[]` for this section of the code. Answers which give more than one answer or which repeat the answer from the previous part will receive zero.

(f) (1 point) Would the code in section 3 still work if `shared[]` were in global memory?

(g) (3 points) Consider the line `y[i] = shared[i*b];` in section 4 of `reduce_kernel()`. Will either of these memory accesses cause a bank conflict? Briefly explain why or why not.

(h) (3 points) Consider the line `y[i] = shared[i*b];` in section 4 of `reduce_kernel()`. Will either of these memory accesses be coalesced? Briefly explain why or why not.

(i) (2 points) Briefly explain why `__syncthreads()` must be called at the start of section 4 of `reduce_kernel()`.

2. (20 points) **Closest Point** Figure 3 provides code implementing a basic version of a closest point calculation, and figure 4 provides a tiled version of the same algorithm. You may assume that `nc` $\gg$ `np` $\gg 1$.

   (a) (3 points) Assume that the data is stored column major. The code currently stores each input data point in `d_points` as a column, so `d_points` is an array with `DIM` rows and `np` columns. A similar layout is used for `d_clusters`. Is this a good choice for arranging the data in `closest_basic_kernel()`, or should we instead have store each point as a row in these arrays? Briefly explain your reasoning.

   (b) (3 points) Write down the line numbers of `closest_basic_kernel()` which contain a floating point operation. Comparisons between floating point values count as a floating point operations, but integer operations and integer comparisons do not count. Multiply-adds which can be fused count as one operation. How many total floating point operations does one thread perform?

   (c) (3 points) Write down the line numbers of `closest_basic_kernel()` which contain a global memory access. How many total global memory accesses does one thread perform?

(d) (1 point) What is the CGMA of `closest_basic_kernel()`? Your answer may depend on `nc`, `np` and/or `DIM`.

(e) (3 points) Write down the line numbers of `closest_tiled_kernel()` which contain a floating point operation. Comparisons between floating point values count as a floating point operations, but integer operations and integer comparisons do not count. Multiply-adds which can be fused count as one operation. How many total floating point operations does one thread perform?

(f) (3 points) Write down the line numbers of `closest_tiled_kernel()` which contain a global memory access. How many total global memory accesses does one thread perform?

(g) (1 point) What is the CGMA of `closest_tiled_kernel()`? Your answer may depend on `nc`, `np` and/or `DIM`.

(h) (3 points) Consider the innermost loop (the loop over `j`) in `closest_tiled_kernel()`. Will this code run faster if `DIM` is 2 or if `DIM` is 3? Briefly explain.

3. (18 points) **Short answer questions.**

   (a) (4 points) **Data Parallelism.** We discussed the fact that GPUs are well suited to data parallel problems, such as convolution or matrix-matrix multiplication. Give **one** architectural reason why GPUs are better suited than traditional CPUs to matrix-matrix multiplication. Are there any conditions under which a distributed message passing architecture (such as we explored using Erlang) would be well-suited to matrix-matrix multiplication? If not, briefly explain why not. If so, specify the condition(s).

   (b) (3 points) **Lots of Threads.** One key to achieving high performance on GPUs is to ensure that kernels have many independent threads: Dozens of threads per core / SP. Give **one** reason why it is beneficial to have so many threads per core in a GPU. Give **one** reason why it is also beneficial to have several threads / processes available for each core of a modern CPU, and **one** reason why it is not necessary to have nearly as many threads / processes for the CPU cores as for the GPU cores to achieve reasonable efficiency. Answers which give more than one of each will receive zero.

(c) (4 points) **Shared Memory.** We ran in to the term "shared memory" twice in the course: once as a type of parallel architecture (as opposed to message passing) and once as a specialized form of storage on the GPUs. Consider the kernel `min_kernel_tree()` in figure 5, which uses the GPU's shared memory. Does it use a shared memory architecture? Briefly explain your answer. Now consider the kernel `min_kernel_atomic()` in figure 5, which does not use the GPU's shared memory. Does it use a shared memory architecture? Briefly explain.

(d) (3 points) **Map-Reduce and Moments.** Assume that we store a very large collection of data values $\{x_i\}_{i=0}^{n-1}$ distributed across a large number of workers in a data center. The values are stored as *(Key1, Value1)* pairs such that *Key1* is i and *Value1* is x_i. Further assume that all of the workers know the expected value of the data set $E[x]$. We would like to compute the central moments $\{\mu_k\}_{k=2}^{k\max}$ where

$$\mu_k \approx \frac{\sum_{i=0}^{n-1}(x_i - E[X])^k}{n}.$$

This calculation is the same problem explored in Homeworks 2 (Erlang) and 5 (CUDA). We would like to store the results as *(Key3, Value3)* pairs where *Key3* is k and *Value3* is μ_k. Is it possible to perform this computation using the Map-Reduce programming pattern (ignore efficiency considerations)?

☐ yes ☐ no (1 point)

If yes, explain what *(Key2, Value2)* representation you would use. If not, briefly explain why not (2 points).

(e) (3 points) **BLAS and Complexity.** You have a computational problem and have managed to write it in two different forms (assume all matrices are $n \times n$ and vectors are of length n):

- A series of n matrix-vector multiplications (Level-2 BLAS).
- A single matrix-matrix multiplication (Level-3 BLAS) plus n vector-vector saxpy operations (Level-1 BLAS).

Assuming that n is small enough that all the matrices and vectors can be stored in the GPU's global memory, which version would run faster? Briefly explain your answer.

(f) (1 point) ☺ Take three, slow, deep breaths. Relax. Write down one sentence that is positive about anything.

4. **Performance Modeling** (10 points) Give one example for how each kind of performance loss listed below can occur in a CUDA program, or give a short (one or two sentence) explanation of why it cannot occur.

   (a) (**2 points**): Communication.

   (b) (**2 points**): Synchronization.

   (c) (**2 points**): Extra computation.

   (d) (**2 points**): Resource contention.

   (e) (**2 points**): Idle processors.

   Hint: feel free to use example from code for other questions on this exam—but if you would rather give your own example that is fine. Your examples should be **short**.

5. **A Different Kind of Tiling** (13 points) Many physical simulation problems are solved using *finite element methods* (FEM). These methods model a physical region using a grid of points. A sequential implementation of FEM involves a sequence of updates of the grid. Each update requires an update of each grid point. We will assume that a single update of a grid-point takes unit time. Thus a sequential implementation uses N time units to update a grid with N-points.

   We will assume that we have P processors, a two-dimensional grid, and that the grid can be divided into P "tiles" of size $K \times K$. Note that this implies that $N = PK^2$. To update a tile, a processor sends four messages, one to each of the neighbouring tiles. Each message has K words. It takes time $\lambda + W$ to exchange messages of W words with a neighbour. Each processor can exchange messages with only one neighbour at a time. You may also assume that computation and communication **cannot** be overlapped. After receiving messages from its 4 neighbours, the processor updates the grid-points for its tile using the sequential approach.

   (a) (**1 point**) What is the time for the sequential algorithm to update a grid of $N = 10,240,000$ points? Hint: this is not a trick question, this is an *easy* question.

   (b) (**1 point**) What is K, if $P = 256$ and $N = 10,240,000$?

   (c) (**2 points**) What is time for the parallel algorithm to update a grid of $N = 10,240,000$ points using $P = 256$ processors? Include both communication and computation time. Assume that $\lambda = 10,000$.

   (d) (**1 point**) What is the speed-up if $N = 10,240,000$, $P = 256$, and $\lambda = 10,000$?

We can amortize the communication cost if we use *overlapping* tiles. Let M be an integer – it's the number of rows or columns that each tile overlaps with each of its neighbouring tiles. To update a tile, each processor sends four messages, one to each of the neigbouring tiles. Each message consists of $M(K + M - 1)$ words. After receiving messages from its 4 neighbours, the processor performs M updates to its tile; thus, one step of the parallel algorithm counts as M steps of the sequential version. For the first of the M updates, the processor works on a tile with $(K+M-1) \times (K+M-1)$ points, for the second iteration, the processor works on a tile with $(K + M - 2) \times (K + M - 2)$ points. For the i^{th} of the M updates, the processor works on a tile with $(K + M - i) \times (K + M - i)$ points.

(e) (**4 points**) What is time for the parallel algorithm to perform $M = 4$ updates to a grid of $N = 10,240,000$ points using $P = 256$ processors? Include both communication and computation time. Assume that $\lambda = 10,000$.

(f) (**2 points**) What is the speed-up if $N = 10,240,000$, $P = 256$, $\lambda = 10,000$, and $M = 4$?

(g) (**2 point**) Does this method of overlapping tiles improve or degrade performance? Give a short explanation for this improvement or degradation in terms of the kinds of overhead described in Question 4, i.e. communication, synchronization, extra computation, resource contention, and/or idle processors.

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 reduce_kernel(float *x, float *y, uint n, uint m) {

  __shared__ float shared[SIZE];

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

  // Section 1.
  for(uint j = 0; j < m; j++)
    shared[j*b + i] = 0.0f;

  // Section 2.
  for(uint k = 0; k < num_elem; k++) {
    uint elem = k * b + i;
    if(elem < n) {
      float curr = x[elem];
      for(uint j = 0; j < m; j++) {
        float temp = shared[j*b + i];
        if(curr > temp) {
          shared[j*b + i] = curr;
          curr = temp;
        }
      }
    }
  }

  // Section 3.
  for(uint stride = b / 2; stride >= 1; stride = stride>>1) {
    __syncthreads();
    if(i < stride) {
      for(uint other_j = 0; other_j < m; other_j++) {
        float curr = shared[other_j*b + i + stride];
        for(uint j = 0; j < m; j++) {
          float temp = shared[j*b + i];
          if(curr > temp) {
            shared[j*b + i] = curr;
            curr = temp;
          }
        }
      }
    }
  }

  // Section 4.
  __syncthreads();
  if(i < m)
    y[i] = shared[i*b];
}
```

Figure 1: CUDA code for the reduction problem for question 1.

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.

```
1  void reduce_cpu(float *x, float *y, uint n, uint m) {
2
3    for(uint j = 0; j < m; j++)
4      y[j] = 0.0f;
5
6    for(uint i = 0; i < n; i++) {
7      float curr = x[i];
8      for(uint j = 0; j < m; j++) {
9        float temp = y[j];
10       if(curr > temp) {
11         y[j] = curr;
12         curr = temp;
13       }
14     }
15   }
16
17 }
```

Figure 2: CPU code which performs the same reduction as the CUDA code in figure 1.

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.

```
// Fortran column major indexing.  We don't actually need the number
// of columns (n), but we'll include it as an argument anyway for when
// we do C-style (row major) indexing.
#define IDX2F(i,j,m,n) (((j)*(m))+(i))

// For each element in d_points, identify the point in d_clusters
// which is closest.  In case of a tie, identify the first.
__global__ void closest_basic_kernel(const float *d_clusters, const uint nc,
                                     const float *d_points, const uint np,
                                     uint *d_closest) {

  // Assign a thread to each element of d_points.
  const uint i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < np) {
    // Initialize with first cluster point.
    float closest_dist2 = 0.0f;
    uint closest_index = 0;
    for(uint d = 0; d < DIM; d++) {
      float diff = (d_points[IDX2F(d,i,np,DIM)] - d_clusters[IDX2F(d,0,nc,DIM)]);
      closest_dist2 += diff * diff;
    }

    // Then consider the remaining cluster points.
    for(uint j = 1; j < nc; j++) {
      float dist2 = 0.0f;
      for(uint d = 0; d < DIM; d++) {
        float diff = (d_points[IDX2F(d,i,np,DIM)]-d_clusters[IDX2F(d,j,nc,DIM)]);
        dist2 += diff * diff;
      }
      if(dist2 < closest_dist2) {
        closest_dist2 = dist2;
        closest_index = j;
      }
    }

    // Write back the result.
    d_closest[i] = closest_index;
  }
}
```

Figure 3: Basic CUDA kernel for the closest point problem for question 2.

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 closest_tiled_kernel(const float *d_clusters, const uint nc,
                                     const float *d_points, const uint np,
                                     uint *d_closest) {

  // Index within the grid.
  const uint ig = blockIdx.x * blockDim.x + threadIdx.x;
  // Index within the block.
  const uint ib = threadIdx.x;

  __shared__ float sh_points[BLOCK_SIZE * DIM];
  if(ig < np)
    for(uint d = 0; d < DIM; d++)
      sh_points[IDX2F(d,ib,BLOCK_SIZE,DIM)] = d_points[IDX2F(d,ig,np,DIM)];

  // Initialize with first cluster point.
  float closest_dist2 = 0.0f;
  uint closest_index = 0;
  for(uint d = 0; d < DIM; d++) {
    float diff = (sh_points[IDX2F(d,ib,np,DIM)] - d_clusters[IDX2F(d,0,nc,DIM)]);
    closest_dist2 += diff * diff;
  }

  __shared__ float sh_clusters[BLOCK_SIZE * DIM];
  for(j_start = 1; j_start < nc; j_start += BLOCK_SIZE) {
    const uint j_last = min(BLOCK_SIZE, nc - j_start);

    __syncthreads();
    if(ib < j_last)
      for(uint d = 0; d < DIM; d++)
        sh_clusters[IDX2F(d,ib,BLOCK_SIZE,DIM)] = d_clusters[IDX2F(d,j_start+ib,np,DIM)];

    __syncthreads();
    for(uint j = 0; j < j_last; j++) {
      float dist2 = 0.0f;
      for(uint d = 0; d < DIM; d++) {
        float diff = (sh_points[IDX2F(d,ib,np,DIM)] - sh_clusters[IDX2F(d,j,nc,DIM)]);
        dist2 += diff * diff;
      }
      if(dist2 < closest_dist2) {
        closest_dist2 = dist2;
        closest_index = j_start + j;
      }
    }
  }

  if(ig < np)
    d_closest[ig] = closest_index;
}
```

Figure 4: Tiled CUDA kernel for the closest point problem for question 2. The macro `IDX2F()` is defined in figure 3.

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.

```
__global__ void min_kernel_tree(int *x, const uint n, int *min) {

  const uint i = threadIdx.x;

  // Allocate shared memory to hold the partial results for this thread.
  __shared__ double sh_min[BS];
  // Initialize with first value assigned to this thread.
  sh_min[i] = x[i];

  // Examine the remaining values assigned to this thread.
  for(uint j = i + GS; j < n; j += GS)
    sh_min[i] = min(x[j],sh_min[i]);

  // Perform the reduction tree.
  for(uint stride = BS >> 1; stride >= 1; stride = stride >> 1) {
    __syncthreads();
    if(i < stride)
      sh_min[i] = min(sh_min[i],sh_min[i + stride]);
  }

  // Copy the final result out of shared memory and back to global memory.
  if(i == 0)
    min[0] = sh_min[0];
}


__global__ void min_kernel_atomic(int *x, const uint n, int *min) {

  const uint i = blockIdx.x * blockDim.x + threadIdx.x;

  // Initialize with first value assigned to this thread.
  int my_min = x[i];

  // Examine the remaining values assigned to this thread.
  for(uint j = i + GS; j < n; j += GS)
    my_min = min(x[j], my_min);

  // Combine results from different threads.
  atomicMin(min, my_min);
}
```

Figure 5: Two kernels to determine the minimum of an array of integers for question 3c.