CS 133 Parallel & Distributed Computing (Spring 2017): Final Exam June 13, 2017

Show all work on this exam. Time: **3 hours**

Please make sure to write your full name and student ID on the exam!

Please make sure that your hand-writing is legible.

Only the notes from the lectures and discussion session are allowed. No other book or electronic device is allowed.

---

1. (45 points, 5 points each) Please give brief (1-2 lines) answers to each of the following questions.
   A. Please identify two major sources of power/energy inefficiency of general-purpose CPU processors.

   B. Comparing the CPU and GPU used for this class, please list three major architecture differences.

   C. Please give two reasons why in some cases FPGAs can be more efficient in performance and/or energy than general-purpose CPU processors.

D. How can two different threads communicate in the Nvidia GPU used in the class? Please discuss all possible cases.

E. Please describe the memory hierarchy of the Xilinx FPGA used in the class project. Please describe two techniques commonly used to improve the memory bandwidth and/or reduce the memory latency.

F. Given a set of website URLs with their text in the list format (URL, text), please use the map-reduce programming paradigm to compute that for each word, the list of all URLs in the sorted order that include that word. Please define the map and reduce functions separately. We assume that a sorting function is available.

G. Suppose that we are going to use the recursive doubling algorithm to implement the MPI Broadcast function on an 8x8 mesh with data located at the bottom-left corner node (0,0). Please list or show the path how the data reaches the node (7,7).

H. In the guest lecture by Dr. Lowney, for supporting task parallel programming, each processor has a local queue and uses work-stealing for load balancing. Why not set up a global task queue for all processors to process tasks?

I. Based on your own experience, please rank the three parallel programming languages that you have learned in the course, OpenMP, MPI, and OpenCL, in terms of (i) ease of learning -- getting started with parallel programming for beginners; and (ii) support for the experienced programmers to get the most efficient parallel implementations.

2. (20 points) In the CNN example used in Labs 3-5, we have input feature maps, weight matrices, and output feature maps (for the simplicity, we don't consider bias). Please answer the following questions.

```
// Sequential CNN implementation
#define NUM 512
#define IMROW 224
#define INIMROW 228
#define KERNEL 5
void CONV(float Cout[NUM][IMROW][IMROW], float Cin[NUM][INIMROW][INIMROW],
          float weight[NUM][NUM][KERNEL][KERNEL])
{
  for(int i=0; i<NUM; i++) {
    for(int h=0; h<IMROW; h++) {
      for(int w=0; w<IMROW; w++)
        Cout[i][h][w] = 0;
    }
  }
  for(int i=0; i<NUM; i++) {
    for(int j=0; j<NUM; j++) {
      for(int h=0; h<IMROW; h++) {
        for(int w=0; w<IMROW; w++) {
          for(int p=0; p<KERNEL; p++) {
            for(int q=0; q<KERNEL; q++)
              Cout[i][h][w] += weight[i][j][p][q]*Cin[j][1*h+p][1*w+q];
          }
        }
      }
    }
  }
}
```

a) For GPU acceleration in Lab 4, please list all data arrays for possible data reuse inside the GPU.

b) If we want to reuse the input feature map data on the GPU as much as possible, how shall we re-structure the program (i.e. via proper loop transformations)? You only have to write down the part of code that you plan to re-structure.

c) For FPGA acceleration in Lab 5, assume that each array only needs to be brought into on-chip BRAM once, please calculate the computation intensity (also called the computation-to-communication ratio) of this problem.

d) For FPGA acceleration in Lab 5, please rank the three data arrays in terms of maximum possible data reuse. Please justify your answer.

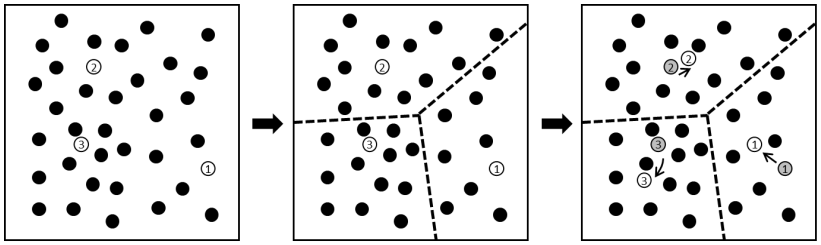
3. (15 points) Given *m* sorted lists, each of length *n*, we assume that all of them fit into main memory and that we can use the shared memory model. We are going to merge them into one sorted list using *p* processors. Considering the parallel merging algorithm based on rank computation as discussed in the class, Alice suggests that we extend it to do an *m*-way merge directly using *p* processors by computing the ranks of each item from one list in the other (m-1) lists using (m-1) binary searches. Bob suggests that we recursively perform 2-way merge using p processors and get a final merged list (assuming that $m=2^t$ for some integer *t*). Please answer the following questions.

    a) Please write down the pseudo code of algorithms used by Alice and Bob.

b) Please analyze the runtime complexity of the approaches proposed by Alice and Bob in terms of m (or t), n, and p (note that we care only the runtime but not the total worktime for this problem). Please show intermediate steps in your calculation.

c) Please compare their runtime and discuss which one has a shorter runtime.

4. (20 points) *k*-means clustering is an algorithm that is widely used for cluster analysis in data mining. *k*-means clustering algorithm partitions N points into *k* clusters in which each point belongs to the cluster whose center (i.e. the mean of the cluster) is closest to the point. We use the following figure to illustrate *k*-means clustering algorithm. It first initializes *k* centers to random locations, and performs clustering to determine which point is belonged to which cluster. After that, we adjust the location of the center for each cluster. The last two steps will be performed several times until the location of centers is fixed.



The following pseudo code is an implementation of *k*-means clustering algorithm. It partitions 16,384 32-dimensional points to 16 clusters. The kernel C code is used for clustering (step 2 in the figure), which will be called repeatedly. It takes a set of points and current center points as inputs, and determines the closest center for each point. Please answer the following questions. Please note that the solutions may not be unique and all reasonable solutions will be given full credit as long as you give a (short and) good justification.

Pseudo Code for k-means clustering

```
#define NUM_POINTS 16384
#define NUM_CENTERS 16
#define DIMS 32

// Omitted code for reading inputs and
// randomly initializing centers

for (int i = 1; i < NUM_ITERATIONS; i++) {
  // Clustering
  kmeans(points, centers, results);

  // Omitted code for resetting centers
  // Count the # of points in a cluster
  int count[NUM_CENTERS] = {0};
  for (int j = 0; j < NUM_POINTS; j++)
    count[results[j]]++;

  // Calculate the mean point of each
  // cluster as the new center
  for (int j = 0; j < NUM_POINTS; j++) {
    for (int k = 0; k < DIMS; k++)
      centers[results[j]][k] +=
        point[j][k];
  }
  for (int j = 0; j < NUM_CENTERS; j++) {
    for (int k = 0; k < DIMS; k++)
      centers[j][k] /= count[j];
  }
}
```

Kernel C code

```
#define NUM_POINTS 16384
#define NUM_CENTERS 16
#define DIMS 32

void kmeans(
  float points[NUM_POINTS][DIMS],
  float centers[NUM_CENTERS][DIMS],
  int res[NUM_POINTS]) {
  for (int i = 0; i < NUM_POINTS; i++) {
    float dis[NUM_CENTERS] = {0};
    for (int j = 0; j < NUM_CENTERS; j++) {
      for (int k = 0; k < DIMS; k++)
        dis[j] += pow(point[i][k]-center[j][k], 2.0);
      dis[j] = pow(dis[j], 0.5);
    }

    float closestDis = INF;
    int closestCenter = -1;
    for (int p = 0; p < NUM_CENTERS; p++) {
      if (dis[p] < closestDis) {
        closestDis = dis[p];
        closestCenter = p;
      }
    }
    // Record the index of the closest center
    // for this point (index i)
    res[i] = closestCenter;
  }
}
```

(a) You want to use OpenCL to accelerate the *k*-means kernel on CS133 server's **CPU**. What global and local work group sizes will you choose, and which loops will you parallelize using work groups in the kernel (just point out the loops without modifying them) to get the maximum speed up?

(b) How do you parallelize the loops by using work groups you mentioned in (a)? You only need to write down the part that you modified in the kernel.

(c) Now, you want to use OpenCL on the **GPU** in cs133 server. Are the loops you wrote in (b) efficient? If not, please explain, and propose a solution for efficient GPU OpenCL implementation. Please specify the size of global/local work groups you want to use, and the corresponding OpenCL code snippet you modified for GPU. (you don't have to write down an entire kernel but just the part you modified.)

(d) We want to accelerate the kernel using **FPGA**. We choose one work group at this time for efficiency, and plan to use loop pipelining. However, this kernel has data dependency so that we cannot achieve fully pipeline. Please 1) point out the data dependency that prevents fully pipeline (not all data dependencies in the kernel), 2) use loop permutation to resolve this dependency, and 3) add a Xilinx OpenCL pipeline attribute or Merlin pipeline pragma to the loop that can achieve II=1.

Xilinx OpenCL pipeline attribute (*__attribute__((xcl_pipeline_loop))*)

Merlin pipeline pragma (#pragma ACCEL pipeline (flatten))