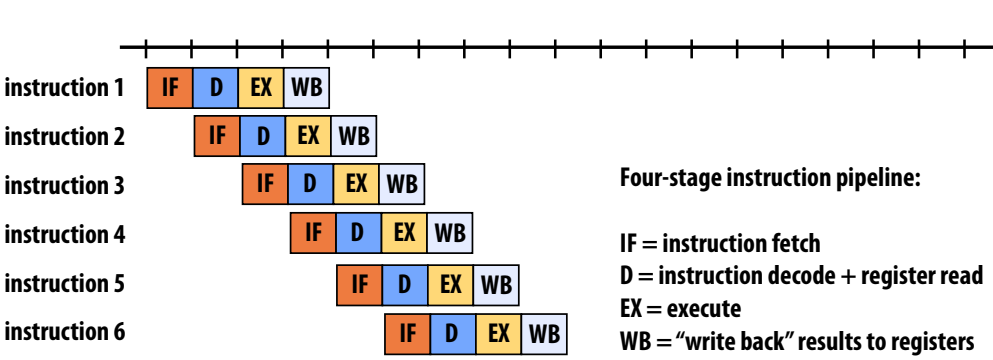
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 2 SOLUTIONS

Problem 1: A Yinzer Processor Pipeline (12 pts)

Yinzer Processors builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. (1 pt) Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

Solution: 1 instruction per clock. One instruction completes each cycle.

B. (1 pt) Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

Solution: 1/4 instruction per clock. One instruction completes every four cycles.

C. (1 pt) What is the latency of completing an instruction?

Solution: 4 cycles

D. (1 pt) Imagine the EX stage is modified to improve its throughput to two instructions per clock. What is the new overall maximum instruction throughput of the processor?

Solution: Throughput remains one instruction per clock. EX will only receive inputs at a rate of one instruction per clock, so its higher throughput ability goes unused.

E. (3 pts) Consider the following C program:

```
float A[100000];
float B[100000];
// assume A is initialized here

for (int i=0; i<100000; i++) {
    float x1 = A[i];
    float x2 = 2*x1;
    float x3 = 3 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop), what is the average instruction throughput of this program? (Hint: You should probably consider at least two loop iterations worth of work).

Solution: The throughput is 4/13 instructions per clock. All instructions within the loop body are dependent on each other, but the last instruction in the body is independent of the first instruction executed in the next iteration of the loop. Therefore there is a steady-state pattern of independent, dependent, dependent, dependent, ...

F. (3 pts) Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

Solution: The main idea is to "unroll the loop four times" and reorder instructions so that each dependent instructions are separated by four other instructions. Therefore, the prior instruction will have completed before the later instruction dependent on it issues. Students might be interested to know the "loop unrolling" is a common compiler optimization. (Bonus fact: given the answer below, can you imagine a reason why setting the ISPC gang size to 16 on an AVX-capable machine might yield better performance than using a gang size of 8?)

```
for (int i=0; i<100000; i+=4) {
   float x1[4], x2[4], x3[4];

   x1[0] = A[i];    x1[1] = A[i+1];   x1[2] = A[i+2];   x1[3] = A[i+3];
   x2[0] = 2*x1[0]; x2[1] = 2*x1[1];  x2[2] = 2*x1[2];  x2[3] = 2*x1[3];
   x3[0] = 3+x2[0]; x3[1] = 3+x2[1];  x3[2] = 3+x2[2];  x3[3] = 3+x2[3];
   B[i] =  x3[0];   B[i+1] = x3[1];   B[i+2] = x3[2];   B[i+3] = x2[3];
}
```

G. (2 pts) Now assume the program is reverted to the original code.

```
// assume iterations of this FOR LOOP are parallelized across multiple
// worker threads in a thread pool.
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this C program, what feature could you add to the **single-core processor** to obtain **peak instruction throughput** (100% utilization of execution resources)? Why? (For example: consider adding SIMD, hardware multi-threading, or a larger cache) You may not change how the instruction pipeline works or how it handles dependent instructions. You may not change the program.

Solution: ***Prof Kayvon botched this question! The handout left out the comment in the code above about assuming the loop iterations are now parallelized over a pool of worker threads. Prof. Kayvon is quite surprized that no-one asked about the question, since as it stands in the original handout the answer is that none of the provided options will help! The answer below is the answer to the question I meant to ask!*** *Hardware multi-threading, with four-threads per core. All instructions within a thread are still dependent (recall one thread would only obtain 4/13 efficiency, like in part B), but the four-cycle latency of instruction execution can be covered by executing instructions from other threads. This is because instructions from different threads will be independent. Also note that adding SIMD to the processor might would not make the unchanged program run faster, since you're told the compiler does not auto-vectorize the binary would not containing SIMD instructions. Even if you assume that the compiler does generate SIMD code (or say, you reimplemented the program in ISPC with a gang size of SIMD-width), although the program would run* ***faster*** *(there are SIMD-width times more execution resources), the* ***efficiency*** *of the program would be the same since there would still be dependencies between the SIMD instructions.*

Problem 2: Parallel Histogram (8 pts)

A sequential algorithm for generating a histogram from the values in a large input array `input` is given below. For each element of the input array, the code uses the function `bin_func` to compute a "bin" the element belongs to (`bin_func` always returns an integer between 0 and `NUM_BINS-1`) and increments the count of elements in that bin.

```
int bin_func(float value);      // external function declaration
float input[N];                 // assume input is initialized and N is a very large

int histogram_bins[NUM_BINS];  // assume bins are initialized to 0

for (int i=0; i<N; i++) {
   histogram_bins[bin_func(input[i])]++;
}
```

You are given a massively parallel machine with N processors (yes, one per input element) and asked by a colleague to produce an efficient parallel histogram routine. To help you out, your colleague hands you a library with a highly optimized parallel sort routine.

```
void sort(int count, int* input, int* output);
```

The library also has the ability to execute a bulk launch of N independent invocations of an application-provided function using the following CUDA-like syntax:

```
my_function<<<N>>>(arg1, arg2, arg3...);
```

For example the following code (assuming `current_id` is a built-in id for the current function invocation) would output:

```
void foo(int* x) {
   printf("Instance %d : %d\n", current_id, x[current_id]);
}

int A[] = {10,20,30}
foo<<<3>>>(A);

"Instance 0 : 10"
"Instance 1 : 20"
"Instance 2 : 30"
```

(question continued on next page)

Using only `sort`, `bin_func` and bulk launch of any function you wish to create, implement a data-parallel version of histogram generation that makes good use of N processors. You may assume that the variable `current_id` is in scope in any function invocation resulting from a bulk launch and provides the number of the current invocation.

```
// External function declarations.  Your solution may or may not use all these functions.
void  sort(int count, int* input, int* output);
int   bin_func(float value);

// input: array of numbers, assume input is initialized
float input[N];

// output: assume all bins are initialized to 0
int   histogram_bins[NUM_BINS];
```

Solution: (Full credit would be given if edge cases for empty bins weren't handled correctly in `compute_counts`.)

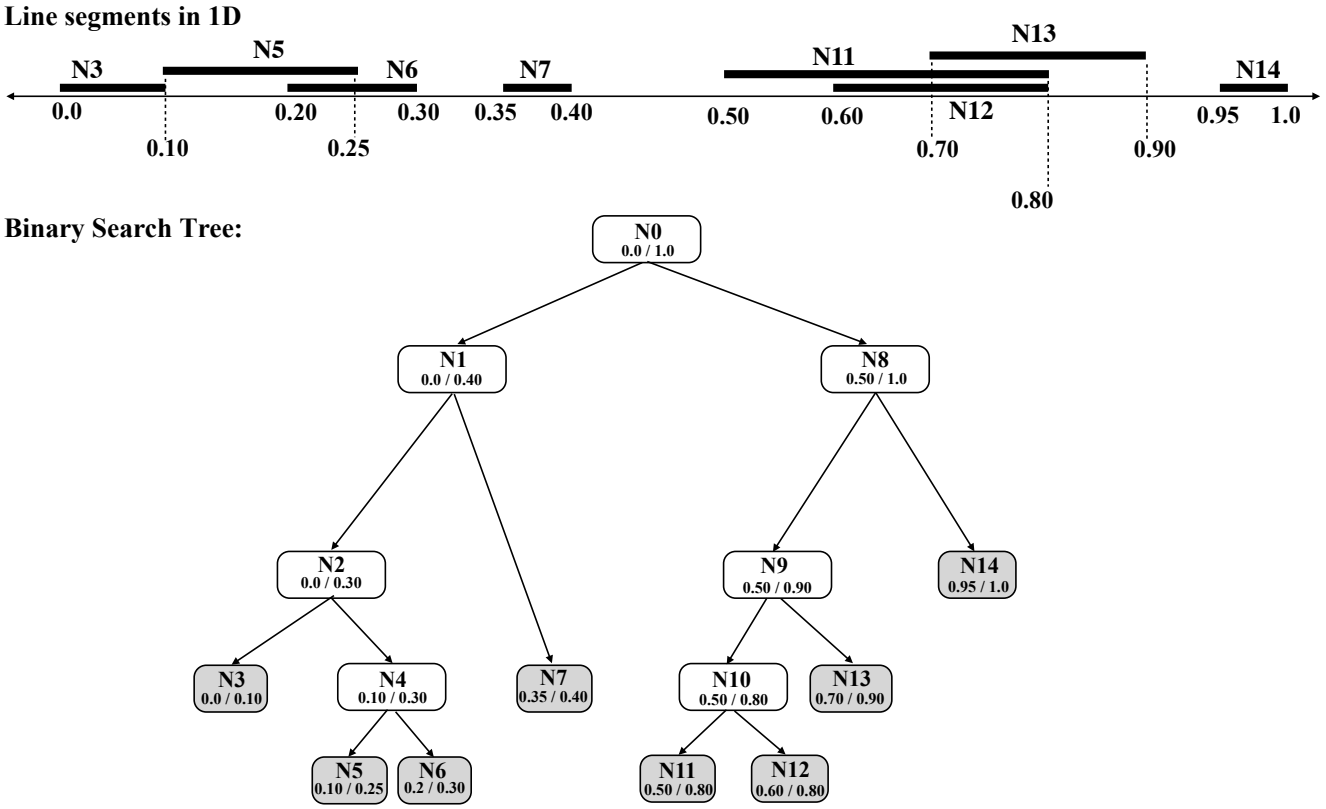
```
int bin_ids[N];
int sorted_bin_idx[N];
int bin_starts[NUM_BINS];  // initialized to -1

void compute_bin(float* input, int* bin_ids) {
   bin_ids[current_id] = bin_func(input[current_id]);
}

void find_starts(int* bin_ids, int* starts) {
   if (current_id == 0 || bin_ids[current_id] != bin_ids[current_id-1])
      starts[bin_ids[current_id]] = current_id;
}

void compute_counts(int* bin_starts, int* histogram_bins, int num_items, int num_bins) {
   if histogram_bins[current_id] == -1) {
      histogram_bins[current_id] = 0;        // edge case for no items in this bin
   } else {
     // find start of next bin.  Tricky edge case: this is not just
     // current_id+1 if next bin in the histogram is empty
     int next_id = current_id+1;
     while(next_id < num_bins && bin_starts[next_id] == -1)
       id++;

     if (next_id < num_bins)
        histogram_bins[current_id] = bin_starts[next_id] - bin_starts[current_id];
     else
        histogram_bins[current_id] = num_items - bin_starts[current_id];
   }
}

int main () {
   // every element computes the bin it falls in
   launch<<<N>>>compute_bin(input, bin_ids);

   // find starting point of each bin in sorted list
   sort(N, bin_ids, sorted_bin_ids);
   launch<<<N>>>find_starts(sorted_bin_ids, bin_starts);

   // compute differences in bin starts to get bin counts
   launch<<<NUM_BINS>>>compute_counts(bin_starts, histogram_bins, N);
}
```

SIMD Tree Search (Extra Challenge Problem)

NOTE: This question is tricky. If you can answer this question you really understand SIMD execution!

The figure below shows a collection of line segments in 1D. It also shows a binary tree data structure organizing the segments into a hierarchy. Leaves of the tree correspond to the line segments. Each interior tree node represents a spatial extent that bounds all its child segments. Notice that sibling leaves can (and do) overlap. Using this data structure, it is possible to answer the question "what is the largest segment that contains a specified point" without testing the point against all segments in the scene.

For example, the answer for point $p = 0.15$ is segment 5 (in node N5). The answer for the point $p = 0.75$ is segment 11 in node N11.




```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};
```

On the following two pages, we provide you two CUDA functions, `find_segment_1` and `find_segment_2` that both compute the same thing: they use the tree structure above to find the id of the largest line segment that contains a given query point.

```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};

// -- computes segment id of the largest segment containing points[threadId]
// -- root_node is the root of the search tree
// -- each CUDA thread processes one query point
__global__ void find_segment_1(float* points, int* results, Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this CUDA thread is searching for
  int threadId = threadIdx.x + blockIdx.x * blockDim.x;
  float p = points[threadId];

  results[threadId] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    while (!node->leaf) {
      // [I-test]: test to see if point is contained within this interior node
      if (p >= node->min && p <= node->max) {
        // [I-hit]: p is within interior node... continue to child nodes
        push(node->right);
        node = node->left;
      } else {
        // [I-miss]: point not contained within node, pop the stack
        if (stack.size() == 0)
          return;
        else
          node = stack.pop();
      }
    }

    // [S-test]: test if point is within segment, and segment is largest seen so far
    if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
       // [S-inside]: mark this segment as ‘‘best-so-far’’
       results[threadId] = node->segment_id;
       max_extent = node->max - node->min;
    }
  }
}
```

```
__global__ void find_segment_2(float* points, int* results, Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this CUDA thread is searching for
  int threadId = threadIdx.x + blockIdx.x * blockDim.x;
  float p = points[threadId];

  results[threadId] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    if (!node->leaf) {
       // [I-test]: test to see if point is contained within interior node
       if (p >= node->min && p <= node->max) {
          // [I-inside]: p is within interior node... continue to child nodes
          push(node->right);
          push(node->left);
       }
    } else {
       // [S-test]: test if point is within segment, and segment is largest seen so far
       if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
         // [S-inside]: mark this segment as ''best-so-far''
         results[threadId] = node->segment_id;
         max_extent = node->max - node->min;
       }
    }
  }
}
```

Begin by studying `find_segment_1`.

Given the input $p = 0.1$, the a single CUDA thread will execute the following sequence of steps: (I-test,N0), (I-hit,N0), (I-test, N1), (I-hit, N1), (I-test, N2), (I-hit, N2) (S-test,N3), (S-hit, N3), (I-test, N4), (I-hit, N4), (S-test, N5), (S-hit, N5), (S-test, N6), (S-test,N7), (I-test, N8), (I-miss, N8). Where each of the above "steps" represents reaching a basic block in the code (see comments):

- (I-test, Nx) represents a point-interior node test against node `x`.
- (I-hit, Nx) represents logic of traversing to the child nodes of node `x` when p is determined to be contained in `x`.
- (I-miss, Nx) represents logic of traversing to sibling/ancestor nodes when the point is not contained within node `x`.
- (S-test, Nx) represents a point-segment (left node) test against the segment represented by node `x`.
- (S-hit, Nx) represents the basic block where a new largest node is found `x`.

The question is on the next page...

A. Confirm you understand the above, then consider the behavior of a **4-wide warp** executing the above two CUDA functions `find_segment_1` and `find_segment_2`. For example, you may wish to consider execution on the following array:

```
points = {0.15, 0.35, 0.75, 0.95}
```

Describe the difference between the traversal approach used in `find_segment_1` and `find_segment_2` in the context of SIMD execution. Your description might want to specifically point out conditions when `find_segment_1` suffers from divergence. (Hint 1: you may want to make a table of four columns, each row is a step by the warp and each column shows each thread's execution. Hint 2: It may help to consider which solution is better in the case of large, heavily unbalanced trees.)

Solution: The main difference between `find_segment_1` and `find_segment_2` is in how they handle the fact that different program instances can reach leaf nodes at different points in time. In `find_segment_1` all program instances in a gang must wait for all other program instances to reach their leaf nodes in order to proceed to the S-test. In `find_segment_2`, program instances are allowed to proceed to the S-test at different times and will do the S-test while the others wait. Because of these differences, `find_segment_2` will perform better on large unbalanced trees. With `find_segment_2`, there will be less divergence than if we had used `find_segment_1` where we will often have to hold up several program instances because at least one program instance in a gang still has not reached a leaf node.

B. Consider a slight change to the code where as soon as a best-so-far line segment is found (inside [S-hit]) the code makes a call to a **very, very expensive function**. Which solution might be preferred in this case? Why?

Solution: Now `find_segment_1` is preferable because the most significant code block now is where the S-test is performed. The amount of work done in the S-test now significantly trumps the amount of work done to get to a leaf node. Therefore, we would rather hold up all program instances until they've all reached their leaf nodes, and then perform the really expensive S-test together to maximize utilization. If we had used `find_segment_2` instead, it would often be the case that some program instances will have reached their leaf nodes while others have not, causing all other program instances to be held up for a really long time while one or a few program instances in the gang perform their S-tests.