15-869 Practice Exercise: Mapping SPMD Programs to SIMD Machines

Question 1:

During class (Lecture 7: Shading Languages) we discussed a number of ways to implement the SPMD abstraction on a processor with explicit vector instructions and vector mask registers. (In our class discussion we worked through how to generate code for an example SPMD program containing nested conditionals.) Describe how you would extend this implementation to execute a SPMD program containing `while` loops with data-dependent loop bounds. To guide your thinking, I've provided some shader code below. The code has a WHILE loop nested within an IF. In both cases, predicates depend on varying values.

Be precise in your thinking: consider how details such as how lane masks should be modified as a result of the IF clause, for each WHILE loop iteration, and after leaving the loop body. Be sure to describe any state (e.g., mask registers) you need to keep around in your implementation. Just like how we drew it up on the board, I'm looking for C-like pseudocode (with vector instructions) of possible compiler output-targeting a SIMD-width=N machine. You can assume the existence of reasonable helper functions to keep your answer short (e.g., `if_any(mask)`, `if_all(mask)`).

```
float mySPMDFunction(int intValue, float floatValueA, float floatValueB)
{
  float tmp;

  if (intValue <= 0)
  {
     tmp = floatValueA;
  }
  else
  {
     tmp = floatValueB;
     int count = intValue;
     while(count > 0)
     {
        tmp *= 2;
        count--;
     }
  }
  return tmp;
}
```

Question 2:

A common operation in computer graphics is to find intersections between a point and scene objects. During the rasterization lecture we discussed algorithms for solving the following instance of this problem: "given a primitive in 2D, find all points contained within the primitive." This question flips the problem around. Given a point and a set of primitives, find all the line primitives that contain the point. To simplify things, we'll carry out the computation in 1D.

Figure 1 (last page) shows a collection of line segments in 1D (the start and end of each segment is given). The figure also shows a binary tree data structure organizing the segments into a spatial hierarchy. Leaves of the tree correspond to the line segments. Each interior node of the hierarchy represents a spatial extent spanned by its children. Notice that sibling leaves can (and do) overlap. Using this data structure, it is possible to answer the question "what segments contain a specified point" without testing the point against all segments in the scene.

The function `find_largest_segment_1` (code is on the next page) uses the tree data structure in Figure 1 to quickly find all line segments containing a point in 1D. It returns the result of `very_expensive_function` called on the *largest* of the line segments containing the point. For example, if this was a simple renderer, one possible implementation of `very_expensive_function` might evaluate a BRDF to compute the color of the line segment at the intersection point. For simplicity in this question, assume that `very_expensive_function` is a straight-line block of code with no conditionals or data-dependent control.

Study the algorithm, and understand how it works. For example, given the input pt_x = 0.1, the algorithm will perform the following sequence of operations: (I-test,N0), (I-hit,N0), (I-test, N1), (I-hit, N1), (I-test, N2), (I-hit, N2) (L-test, N3), (VEF, N3), (I-test, N4), (I-hit, N4), (L-test, N5), (VEF, N5), (L-test, N6), (L-test,N7), (I-test, N8), (I-miss, N8)

where:

- (I-test, Nx) represents a point-interior node test against Node X.
- (I-hit, Nx) represents logic of traversing to the child nodes after it is determined the query point is contained within Node X.
- (I-miss, Nx) represents logic of traversing to sibling/ancestor nodes when the point is not contained within node X.
- (L-test, Nx) represents a point-leaf node test against the segment represented by Node X.
- (VEF, Nx) represents `very_expensive_function` executed on node X.

Now consider simultaneous SIMT-style execution of `find_largest_segment_1` on a 4-wide system using the four points 0.1, 0.4, 0.7, and 0.75 as inputs. Using the notation established above, chart the utilization of each vector "lane" of the processor in the 4-column matrix below (columns indicate behavior of each of the four SIMT lanes, and rows correspond to processor behavior at a particular point in time). Note that the first column of the matrix should contain the values given in the example for point 0.1 above. It may be helpful to use "---" to indicate that a lane's operation is masked at a particular time.

```
struct Node {
   float min, max;
   bool leaf;
   Node* left;
   Node* right;
};

// returns the value of very_expensive_function(node, pt_x) for the largest
// segment containing pt_x.  If no segment contains pt_x, returns NO_SEGMENT
float find_largest_segment_1(float pt_x, Node* root_node)
{

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;
  float result = 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 interior node
      if (pt_x >= node->min && pt_x <= node->max)
      {
        // I-hit: continue to child nodes
        push(node->right);
        node = node->left;
      }
      else
      {
        // I-miss: point not contained within node
        if (stack.size() == 0)
          return result;
        else
          node = stack.pop();
      }
    }

    // L-test: test to see if point is contained within line segment (leaf node)
    if (pt_x >= node->min && pt_x <= node->max && (node->max-node->min) > max_extent)
    {
       // this basic block is referred to as VEF in problem description:
       result = very_expensive_function(node, pt_x);
       max_extent = node->max – node->min;
    }
  }

  return result;
}
```

Tip: if any row of the matrix below indicates two lanes are performing different work, you might want to check your thinking. Why?

Step	pt_x = 0.1	pt_x = 0.4	pt_x = 0.7	pt_x = 0.75
1	(I-test, N0)	(I-test, N0)	(I-test, N0)	(I-test, N0)
2	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				

Question 3:

The function `find_largest_segment_2` produces the same output as `find_largest_segment_1`, but the implementation is different. Chart the code's SIMT execution behavior on the same four rays as in question 2.

```
float find_largest_segment_2(float pt_x, Node* root_node)
{

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;
  float result = 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 (pt_x >= node->min && pt_x <= node->max)
       {
          // I-hit: continue to child nodes
          push(node->right);
          push(node->left);
       }
    }
    else
    {
       // L-test: test to see if point is contained within line segment (leaf node)
       if (pt_x >= node->min && pt_x <= node->max && (node->max-node->min) > max_extent)
       {
          // this basic block is referred to as VEF in the problem description:
          result = very_expensive_function(node, pt_x);
          max_extent = node->max – node->min;
       }
    }
  }

  return result;
}
```

Hint: if any row of the matrix below indicates two lanes are performing different work, you might want to check your thinking. Why?

Step	pt_x = 0.1	pt_x = 0.4	pt_x = 0.7	pt_x = 0.75
1	(I-test, N0)	(I-test, N0)	(I-test, N0)	(I-test, N0)
2	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)	(I-hit, N0)
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				

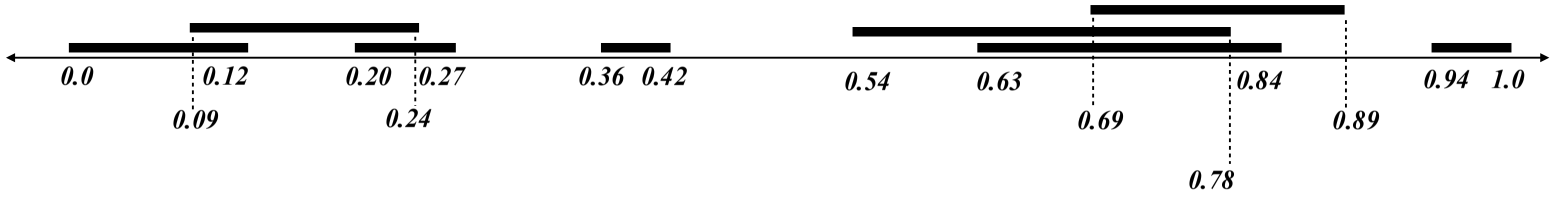
Question 4:

1. What are the advantages and disadvantages of the two implementations given in questions 2 and 3? Although I only provided one example dataset of segments and point queries in this assignment, in forming your answer I suggest you consider the behavior of these functions under varying characteristics of the binary tree (hint: consider very large, unbalanced trees), different costs of `very_expensive_function`, or different point queries. In what situations would you prefer `find_largest_segment_1`? What about `find_largest_segment_2`?

2. What is a simple change to the code that would not change the final result, but significantly reduce both execution divergence and total work performed by the algorithm?

Figure 1

Line segments in 1D



Corresponding binary tree