Stanford CS149: Parallel Computing
Written Assignment 3
Due October 26th

Problem 1: Fusion, Fusion, Fusion

Your boss asks you to buy a computer for running the program below. The program uses a math library (`cs149_math`). The library functions should be self-explanatory, but example implementations of the `cs149math_add` and `cs149math_sum` functions are given below.

```
const int N = 10000000;   // very large

void  cs149math_sub(float* A, float* B, float* output);
void  cs149math_mul(float* A, float* B, float* output);

void cs149math_add(float* A, float* B, float* output) {
  // Recall from written asst 1 that this OpenMP directive tells the
  // C compiler that iterations of the for loop are independent, and
  // that implementations of C compilers that support
  // OpenMP will parallelize this loop using multiple threads.
  #omp parallel for
  for (int i=0; i<N; i++)
    output[i] = A[i]+B[i];
}

float cs149math_sum(float* A) {     // compute sum of all elements of the input array
  atomic<float> x = 0.0;
  #omp parallel for
  for (int i=0; i<N; i++)
    x += A[i];
  return x;
}

/////////////////////////////////////////////////////////////////
// The program is below:
/////////////////////////////////////////////////////////////////

// assume arrays are allocated and initialized
float* src1, *src2, *src3, *tmp1, *tmp2, *tmp3, *dst;

cs149math_add(src1, src2, tmp1);     // 1
cs149math_mul(tmp1, src3, tmp2);     // 2
cs149math_mul(tmp2, src1, tmp3);     // 3
float x = cs149math_sum(tmp2) / N;   // 4
if (x > 10.0) {
  cs149math_mul(tmp3, src1, tmp1);   // 5
  cs149math_add(src1, tmp1, tmp2);   // 6
  cs149math_add(src1, tmp2, dst);    // 7
} else {
  cs149math_add(tmp3, src2, tmp1);   // 8
  cs149math_mul(src2, tmp1, tmp2);   // 9
  cs149math_mul(src2, tmp2, dst);    // 10
}
```

The question is on the next page...

You have two computers to choose from, of equal price. (Assume that both machines have the same 16MB cache and 0 memory latency.)

1. Computer A: Four cores 1 GHz, 4-wide SIMD, 192 GB/sec bandwidth
2. Computer B: Four cores 1 GHz, 8-wide SIMD, 128 GB/sec bandwidth

ASSUME THAT YOU ARE ALLOWED TO REWRITE THE CODE, INCLUDING REPLACE LIBRARY CALLS IF DESIRED, (provided that it computes exactly the same answer—You can parallelize across cores, vectorize, reorder loops, etc. but you are not permitted to change the math operations to turn adds into multiplies, eliminate common subexpressions etc.). **Please give the arithmetic intensity of your new program assuming that both loads and stores are 4 bytes of data transfer. (You can also assume 1 GB is** 10^9 **bytes.)** As a result, which machine do you choose? Why? (If you decide to change the program please give a pseudocode description of your changes. What is parallelized, vectorized, what does the loop structure look like, etc.)

Problem 2: Misc Problems

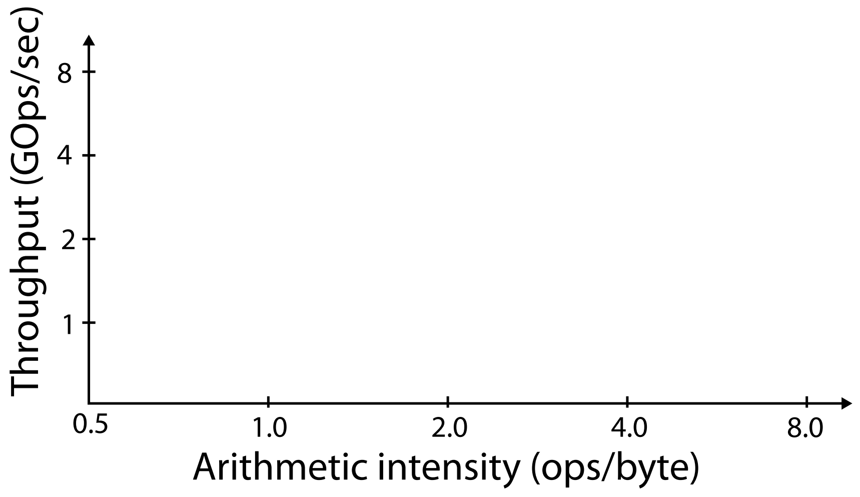
A. A key idea in this course is the difference between *abstraction* and *implementation*. Consider two abstractions we've studied: ISPC's `foreach` and Cilk's spawn construct. **Briefly describe how these two abstractions have similar semantics.** (Hint: what do the constructs declare about the associated loop iterations? Be precise!). **Then briefly describe how their implementations are quite different** (Hint: consider their mapping to modern CPUs). As a reminder, we give you two syntax examples below:

```
ISPC foreach:                          Cilk:
============================           ============================
                                       void f(int i, float* x, float* y) {
                                         x[i] = y[i]
                                       }

foreach (i = 0 ... 100) {              for (int i=0; i<100; i++) {
  x[i] = y[i];                           cilk_spawn f(i, x, y);
}                                      }
```

B. In class we described the usefulness of making roofline graphs, which plots the instruction throughput of a machine (gigaops/sec) as a function of a program's arithmetic intensity (ops performed per byte transferred from memory). Consider the roofline plot below. Please plot the roofline curve for a machine featuring a **1 GHz dual-core processor. Each core can execute one 4-wide SIMD instruction per clock**. This processor is connected to a memory system providing 4 GB/sec of bandwidth. *Hint: what is the peak throughput of this processor? What are its bandwidth requirements when running a piece of code with a specified arithmetic intensity?*

Plot the expected throughput of the processor when running code at each arithmetic intensity on the X axis, and draw a line between the points.



Problem 3: Data-Parallel Thinking

Assume you are given a library that can 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 would output: (`id` is a built-in id for the current function invocation)

```
void foo(int* x) {
   printf("Instance %d : %d\n", id, x[id]);
}
int A[] = {10,20,30}
foo<<<3>>>(A);

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

The library also provides the data-parallel function `exclusive_scan` (using the + operator) that works as discussed in class.

```
exclusive_scan(N, in, out);

Example usage:
N    = 6
in   = {1, 2, 3, 4, 5, 6}
=====================================
out  = {0, 1, 3, 6, 10, 15}
```

In this problem, we'd like you to design a data-parallel implementation of `largest_segment_size()`, which, given an array of flags that denotes a partitioning of an array into segments, computes the size of the longest segment in the array.

```
int largest_segment_size(int N, int* flags);
```

The function takes as input an array of N flags (`flags`) (with 1's denoting the start of segments), and returns the size of the largest segment. The first element of flags will always be 1. For example, the following flags array describes five segments of lengths 4, 2, 2, 1, and 1.

```
N       = 10
flags   = {1,  0,  0,  0,  1,  0, 1,  0, 1,  1}
=============================================
result:  = 4
```

Questions on next page...

A. The first step in your implementation should be to compute the size of each segment. Please use the provided library functions (bulk launch of a function of your choice + `exclusive_scan` to implement the function `segment_sizes()` below. *Hint: We recommend that you get a basic solution done first, then consider the edge cases like how to compute the size of the last segment.*

```
// Example output of segment_sizes(N, flags, num_segs, sizes):
//   N        =  8
//   flags    =  {1, 0, 1, 0, 0, 0, 1, 0}
//   ==================================
//   num_segs = 3
//   sizes    = {2, 4, 2}


// you may wish to define functions used in bulk launches here
```

```
// You can allocate any required intermediate arrays in this function
// You may assume that 'seg_sizes' is pre-allocated to hold N elements,
// which is enough storage for the worse case where the flags array
// is all 1's.
void segment_sizes(int N, int* flags, int* num_segs, int* seg_sizes) {

}
```

B. Now implement `largest_segment_size()` using `segment_sizes()` as a subroutine. **NOTE: this problem can be answered even without a valid answer to Part A.** Your implementation may assume that the number of segments described by `flags` is always a power of two. A full credit implementation will maximize parallelism and minimize work when computing the maximum segment size from an array of segment sizes. *Hint: we are looking for solutions with `lg2(num_segs)` span.*

```
// you may want to implement helper functions here that are called via bulk launch
```

```
int largest_segment_size(int N, int* flags) {

  int num_segs;
  int seg_sizes[N];
  segment_sizes(N, flags, &num_segs, seg_sizes);

}
```