Stanford CS149: Parallel Computing
Written Assignment 2

Problem 1: Effects of Arithmetic Intensity (25 points)

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 an example implementation of the `cs149math_add` function is given below.

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

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 across CPU cores.
  #omp parallel for
  for (int i=0; i<N; i++)
    output[i] = A[i]+B[i];
}
void cs149math_sub(float* A, float* B, float* output) { ... }
void cs149math_mul(float* A, float* B, float* output) { ... }

////////////////////////////////////////////////////////////////

float* A, *B, *C, *tmp1, *tmp2, *result;  // assume arrays are allocated and initialized

cs149math_add(A, B, tmp1);          // 1
cs149math_mul(tmp1, C, tmp2);       // 2
cs149math_mul(tmp2, A, tmp1);       // 3
cs149math_add(A, tmp1, tmp2);       // 4
cs149math_mul(B, tmp2, tmp1);       // 5
cs149math_mul(B, tmp1, tmp2);       // 6
cs149math_mul(B, tmp2, tmp1);       // 7
cs149math_sub(C, tmp1, result);     // 8
```

You have two computers to choose from, of equal price. (Assume that both machines have the same 1MB cache and 0 memory latency.)

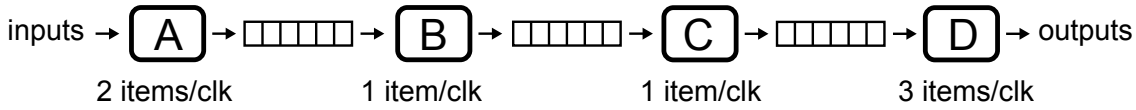
1. Computer A: Four cores 1 GHz, 32-wide SIMD, 128 GB/sec bandwidth
2. Computer B: Eight cores 1 Ghz, 8-wide SIMD, 256 GB/sec bandwidth

ASSUME THAT YOU ARE ALLOWED TO REWRITE THE PROBLEM, INCLUDING THE LIBRARY 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.), which machine do you choose? Why? (If you decide to change the program please give a rough description of your changes. What is parallelized, vectorized, what does the loop structure look like, etc.) Hint: begin with a computation of the current program's arithmetic intensity.

Additional space for your answer...

Problem 2: Understanding Pipelines (25 pts)

Consider the four-stage pipeline below. Each stage in the pipeline receives elements from a 6-element input queue, and can process elements from its input queue at the rates shown in the figure. The behavior of each stage generates one output element for each input element. **A stage stalls (does no work in a clock) if there is no room in the output queue to place a result.** You can assume that inputs arrive at stage A infinitely fast, so stage A will always process two items/clk whenever it can.



For example, consider the following behavior of the pipeline:

- t=0: Stage A processes two elements from its input queue, emits two elements to the Stage B input queue (size=2)
- t=1: Stage B processes one element from its input queue and emits one element to the Stage C input queue (size=1). Stage A processes two new elements from its input queue, emits two elements to the Stage B input queue (size=??).
- t=2: Stage C processes one element from its input queue, emits one element to its output, B processes one elements from it's input queue, A processes two elements from it's input queue, ...

What is the throughput of the pipeline in terms of completed items/clock? What is start-to-end latency of the entire pipeline processing one element? (define latency as the time between the clock where an element is first processed by A and the clock that D finishes processing it. Do not consider time spent waiting in the queue before A) **(Be careful: In both cases, make sure you give answers for the steady-state behavior of the pipeline (including its queues), not its initial startup.)**

Problem 3: Parallel Histogram (25 pts)

Consider the following code which computes a histogram for all values in the array `A`. Assume the code is run by T threads. Throughout this problem, assume that `N` is a very large value.

```
int A[N];  // contains values between 0 and 16

// assume all bins are initialized to 0
int bins[16];

// the following code is run by each thread. Assume thread_id
// takes on a value between 0 and T-1.  You can assume T divides N.

int elements_per_thread = N/T;
int start = thread_id * elements_per_thread;
int end = start + elements_per_thread;

for (int i=start; i<end; i++)  // assume loop management has 0 cost
   bins[A[i]]++;
```

A. (7 pts) There is a correctness bug in this code. Assuming you only have the single synchronization primitive `atomicAdd(int* x, int amount)` please fix the bug in the code.

B. (8 pts) Assume that all loop body operations (loads, stores, adds) take 1 cycle, and ignore any costs of loop iteration. How many cycles does the body of the ORIGINAL UNMODIFIED loop take? (be careful, there is one integer add in the loop, but how many loads and stores are there?)

If we assume that an `atomicAdd` operation (the load/add/store) takes 7 cycles, what is the speedup of your solution from part A **compared to a single-threaded version of the original code**, assuming your part B solution is run using four threads on four cores?

C. (10 pts) Assume that you are also given a `barrier()` as a synchronization primitive. How would you modify the code to improve the speedup. Hint: consider a common trick in this class for removing fine-grained synchronization.

Problem 4: Where Did the Speedup Go (25 pts)

You want to determine all prime numbers up to 10,000,000, using OpenMP to exploit multicore parallelism. The following function tests whether a number x is prime. It exploits the property that any composite number x must be divisible by some number y, such that $1 < y \leq \sqrt{x}$:

```
bool test_prime(int x)
{
    if (x == 0)
        return false;
    int lim = (int) sqrt((double) x);
    for (int i = 2; i <= lim; i++) {
        if (x % i == 0)
            return false;
    }
    return true;
}
```

Your overall scheme is to use OpenMP to launch T threads:

```
// this code parallelized nthreads iterations of the loop onto
#pragma omp parallel for schedule(static)
for (int t = 0; t < nthreads; t++) {
  tf(t, nthreads, xmax, isprime);
}
```

where each instance of thread function `tf` will do the primality testing for some subset of the possible values, setting the elements of a global array `isprime` to either `true` or `false`.

Here are two possible thread functions:

```
void thread_run_interleave(int t, int nthreads, int xmax, bool *isprime) {
  for (int x = t; x <= xmax; x += nthreads) {
        isprime[x] = test_prime(x);
  }
}

void thread_run_chunk(int t, int nthreads, int xmax, bool *isprime) {
  int npt = (xmax + nthreads - 1)/nthreads;
  int xstart = npt*t;
  int xlast = t == nthreads-1 ? xmax : xstart + npt - 1;
  for (int x = xstart; x <= xlast; x++) {
        isprime[x] = test_prime(x);
  }
}
```

Using these different functions, the following speedups are achieved:

Threads	2	3	4	5	6
interleave	1.04	1.93	1.92	3.53	1.93
chunk	1.62	2.17	2.74	3.08	3.86

A. Explain the speedup results for `thread_run_interleave`: **(Note: answering this question requires thinking about the behavior of `test_prime`. e.g., when must it do very little work?)**

(a) (8 pts) Why is their no speedup for 2 threads?

(b) (9 pts) Why isn't the speedup monotonic with respect to the number of threads? (Hint: note behavior at 4 and 6 threads.)

B. (8) Explain why `thread_run_chunk` gets some speedup, but not very much?