Stanford CS149: Parallel Computing
Written Assignment 3

Problem 1: Independent vs. Concurrent (15 pts)

Both ISPC and CUDA feature the concept of a "bulk work launch".

- In ISPC we launched a set of N tasks using `launch [] myISPCFunction();` (Keep in mind each task is executed by a gang of ISPC program instances).
- In CUDA we launched a set of CUDA thread blocks using `myCudaFunction<<<N>>>();`

In both cases, we talked about how the *implementation* of these operations, regardless of the value of N (which can be arbitrarily large), used a shared work queue with N items and a worker pool of a fixed number of workers. In the case of ISPC, workers were CPU threads. In the case of CUDA, workers are GPU cores. These workers would run until all work for the launch was complete, continually grabbing the next item in the work queue, **processing the corresponding task/thread block to completion**, then returning to the work queue to get the next task.

Neither ISPC nor CUDA provide a synchronization construct that is a barrier across *all program instances in all tasks from a single launch*, or in the case of CUDA, *across all CUDA threads in all thread blocks from a single launch*. (You may remember that CUDA `__syncthreads()`'s is only a barrier across all threads in the *same* thread block.) Consider what it would take to implement such an operation. Please describe why the current implementation of these systems would have to change significantly if they were to correctly support such an operation. (Hint: we bolded "process to completion" above for a reason.)

Problem 2: Memory Coherence and Synchronization (25 pts)

Consider the following sequence of events that occur as processors 1 and 2 in a cache-coherent multiprocessor system seek to write to the same address X. Assume the system's interconnect is a shared bus and coherence is maintained using an invalidation-based protocol like MSI. **Also assume that the line is in the invalid state at the start of this problem.**

1. Processor 1 issues store instruction to X
2. Cache 1 is searched for cache line containing X, a cache miss occurs
3. Cache 1 places BusRdX (for the line containing X) command on bus
4. Other caches perform snooping and invalidate their copy of the line
5. Cache 1 loads the line from memory into the M state.
6. Cache 1 updates the new value at address X
7. Cache 2 issues a command BusRd
8. Cache 1 snoops the request, flushes the line containing X to memory, then changes its copy of the line to the S state.
9. Cache 2 loads the line containing X from memory into the S state.

A. (10 pts) In class we talked about how from the perspective of memory coherence, the write by P1 **commits** at step 4. (in other words, this commit point is the point as which all processors can agree the write occurs.) Describe why this is the case, since the data for the write is not written into the cache until step 5, and not flushed to memory until step 8.

B. (15 pts) **(This question is unrelated to the previous parts of this problem.)** Consider executing the following code on a multi-core CPU with invalidation-based coherence. The code computes the maximum of a sequence of integers in parallel. **(Assume the machine has infinite memory bandwidth and that the overheads of work distribution in OpenMP is negligible.)**

```
int max_value = -1;
int N = 5000000;
int input[N];  // initialized to random integers (chosen from a uniform
               // distribution) between 0 and 250

#omp parallel for schedule(dynamic)
for (int i=0; i<N; i++) {
    atomic_max(&max_value, input[i]);   // atomically store the result of
                                        // max(max_value, input[i]) into &max_value
                                        // consider this operation as a write in
                                        // the coherence protocol, regardless of the outcome
                                        // of the comparison
}
```

Your co-worker takes a look at the code and suggests the following simple change:

```
#omp parallel for schedule(dynamic)
for (int i=0; i<N; i++) {
  if (input[i] > max_value)
    atomic_max(&max_value, input[i]);
}
```

Your boss hears this and yells, "That is only going to hurt performance since you are just duplicating the comparison in the `atomic_max`!" However, your colleague ignores the boss, runs the code, and it runs significantly faster. Carefully consider the workload, then state why the optimization helps performance. (Hint: the optimization helps the most towards the end of the program's execution. Why?)

Problem 3: Parallel Histogram Generation (30 pts)

Your friend implements the following parallel code for generating a histogram from the values in a large input array `input`. 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 a count of elements in that bin. Their port targets a small parallel machine with only two processors. *This machine features 64-byte cache lines and uses an invalidation-based cache coherence protocol.* Your friend's implementation is given below.

```
float input[N];                  // assume input is initialized and N is a very large
int   histogram_bins[NUM_BINS];  // output bins
int   partial_bins[2][NUM_BINS]; // assume bins are initialized to 0
                                 // assume partial_bins is 64-byte aligned

////////////////////////// Code executed by thread 0 //////////////////////////
for (int i=0; i<N/2; i++)
   partial_bins[0][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point

for (int i=0; i<NUM_BINS; i++)
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];

////////////////////////// Code executed by thread 1 //////////////////////////
for (int i=N/2; i<N; i++)
   partial_bins[1][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point
```

A. (10 pts) Your friend runs this code on an input of eight million elements (`N`=8,000,000) to create a histogram with four bins (`NUM_BINS`=4). They are shocked when their program obtains far less than a linear speedup, and they glumly assert they needs to completely restructure the code to eliminate load imbalance. You take a look and recommend that they not do any coding at all, and just create a histogram with 16 bins instead. Who's approach will lead to better parallel performance? Why?

B. (10 pts) Inspired by their new-found great performance, your friend concludes that more bins is better. They try to use the provided code from part A to compute a histogram of N=20,000 elements with 10,000 bins. They are shocked when the speedup obtained by the code drops. Improve the existing code to scale near linearly with the larger number of bins. (Please provide pseudocode as part of your answer – it need not be compilable C code.)

C. (10 pts) Your friend changes `bin_func` to a function with *extremely high arithmetic intensity*. (The new function requires 100000's of instructions to compute the output bin for each input element). If the histogram code **provided in part A** is used with this new `bin_func` do you expect scaling to be better, worse, or the same as the scaling you observed using the old `bin_func` in part A? Why? (Please ignore any changes you made to the code in part B for this question.)

Problem 4: Running a CUDA Program on a GPU (30 pts)

Consider a NVIDIA GPU with the following specs. The GPU runs CUDA programs in the manner discussed in class.

- The processor has sixteen cores running at 1 GHz.
- Each core provides execution contexts for up to 1024 CUDA threads. Like the GPUs discussed in class, once a CUDA thread is assigned to an execution context, the processor runs the thread to completion before assigning a new CUDA thread to the context.
- The cores execute CUDA threads in a SIMD fashion running 32 consecutively numbered CUDA threads together using the same instruction stream (32-wide "warps").
- The cores will fetch/decode one single-precision arithmetic instruction (add, multiply, etc.) per clock. Keep in mind this instruction is executed on an entire warp's worth of threads in that clock.
- The GPU has infinite memory bandwidth and zero memory latency.

A. (5 pts) When running at peak utilization. What is the processor's **maximum throughput** of single-precision **math operations**? (In your answer, please consider one multiply of two single-precision numbers as one "operation".)

B. (10 pts) Consider a single CUDA kernel launch that executes the following CUDA kernel on the processor. In this program each CUDA thread computes one element of the output array Y. **Assume the program is compiled using a CUDA thread-block size of 1024 threads, and that the host code launches enough thread blocks so that there is exactly one CUDA thread per output array element.**

```
__global__ void foo(float* X, float* Y) {

   // get array index from CUDA block/thread id
   // blockDim.x = 1024
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   float tmp = X[idx];
   for (int i=0; i<1000; i++) {
       tmp += 1.f;
   }
   Y[idx] = tmp;
}
```

Assuming the input and output arrays have size 4x1024=4096 elements, will the program realize peak performance on the GPU? Why or why not?

C. (5 pts) Now consider the same setup as in part B, but with input/output array size = 1024x1024 elements. Do you now expect the program to realize peak performance on the GPU? Why or why not?

D. (10 pts) Now imagine the program is changed to the following. Assuming the input/output array size = 1024x1024 elements, will the program realize peak performance on the GPU? Why or why not? **Keep in mind, the input array X is initialized as `X[] = {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}`.**

```
__global__ void foo(float* X, float* Y) {

   // get array index from CUDA block/thread id
   // blockDim.x = 1024
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   float tmp = X[idx];
   for (int i=0; i<X[idx]/32; i++) {
       tmp += 1.f;
   }
   Y[idx] = tmp;
}
```