Stanford CS149: Parallel Computing
Written Assignment 3

Problem 1: A Lock (15 pts)

A. (15 pts) Recall a basic test and test-and-set lock, written below using compare and swap (`atomicCAS`) Keep in mind that `atomicCAS` is atomic although it is written in C-code below.

```
int atomicCAS(int* addr, int compare, int val) {
  int old = *addr;
  *addr = (old == compare) ? val : old;
  return old;
}

void LOCK(int* lock) {
   while (1) {
      if (*lock == 0)
         if (atomicCAS(lock, 0, 1) == 0)
            return;
   }
}
```

Imagine a program that uses this lock to synchronize access to a variable shared by 32 threads. This program is run on a processor with **four cores**, each of which are **eight-way multi-threaded** (32 total execution contexts across the machine.) You can assume that **the lock is highly contended, the cost of lock acquisition on lock release is insignificant and that the size of the critical section is large (say, 100,000's of cycles)**. You can also assume there are no other programs running on the machine.

Your friend (correctly) points there is a performance issue with your implementation and it might be advisable to not use a spin lock in this case, and instead use a lock that de-schedules a thread off an execution context instead of busy waiting. You look at your friend, puzzled, mentioning that the test-and-test-and-set lock means all the waiting threads are just spinning on a local copy of the lock in their cache, so they generate no memory traffic. What is the problem they are referring to? **(A full credit answer will mention a fix that specifically mentions what waiting threads to deschedule.)**

Problem 2: Running a CUDA Program on a GPU (40 pts)

The CS148 course staff decides to dip their toes into the GPU design design business and spend their summer creating a new GPU, which, given their poor marketing sense, they call a PKPU (for Prof. Kayvon Processing Unit, or Prof Kunle Processing Unit). The processor runs CUDA programs exactly the same manner as the NVIDIA GPUs discussed in class, but it has the following characteristics:

- The processor has eight cores running at 1 GHz.
- Each core provides execution contexts for up to 128 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 threads in an implicit SIMD fashion running 16 consecutively numbered CUDA threads together using the same instruction stream (the PKPU implements 16-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 in that clock.
- All CUDA thread blocks on a single core cannot exceed 16 KB of shared memory storage.

A. (8 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".)

Consider a CUDA kernel launch that executes the following CUDA kernel on the processor. In this program each CUDA thread computes one element of the results array `Y` using 1000 elements from the input array `X` as input. **Assume the program is compiled using a CUDA thread-block size of 128 threads, and that enough thread blocks are created so 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
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int input_idx = 1000 * idx;

   float result = 0.f;

   for (int i=0; i<1000; i++) {                  // 0 cycles (ignore arithmetic here)

      float val = X[input_idx+i];                // memory load (ignore arithmetic here)

      if (((int)val / 1000) % 2 == 0)            // 2 arithmetic cycles
         result += doA(val);                     // 7 arithmetic cycles
      else
         result += doB(val);                     // 7 arithmetic cycles
   }

   Y[idx] = result;                              // memory store

}
```

B. (8 pts) The TAs hook the PKPU up to a memory system that provides **32 GB/sec of bandwidth** and has a **memory request latency of 100 cycles**. They run the code on a 128,000-element input array initialized as `X[] = {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}`. *The output array `Y[]` is only 128 elements... hint: how many thread blocks is this?*. The TA's observe that the program *does not* realize peak performance on the PKPU. Mario is devastated. What is the primary reason for the low performance? (Please assume the input and output arrays are resident in GPU memory at the time of the kernel launch. Transfer between host and GPU memory is not relevant in this problem.)

C. (8 pts) Juan steps in and says, "hey, let me take a look", and runs the same program on an output arrays of size 128×1024 elements and 128×1024×1024 elements. Does he observe significantly different *processing throughput* from the PKPU on the two workloads? Why or why not? (Please assume both the small and large workloads fit comfortably in GPU memory.)

D. (8 pts) Mario looks at the timing of Juan's experiments on the largest dataset and says "Juan, this program is not achieving peak utilization of the processor's execution units." What is the problem limiting performance? What percentage of peak PKPU throughput does Mario observe? Remember:

- The processor runs a instruction on an entire warp worth of CUDA threads in a single cycle.
- The memory system provides 32 GB/sec of bandwidth with a request latency of 100 cycles. (When a warp issues a load instruction the data for all threads will return in 100 cycles.)

E. (8 pts) After some intense discussion, the TAs hook a new memory system up to the PKPU that provides **64 GB/sec of bandwidth** (still with a request latency of 100 cycles). Juan also **rewrites the program to interleave elements of the input array so that they are stored in the following order:**

```
    0...  999
 2000... 2999
 4000... 4999
     ...
30000...30999
 1000... 1999
 3000... 3999
 5000... 5999
     ...
31000...31999
     ...
32000...32999
34000...34999
     ...
```

The TA's congratulate themselves on a job well done and head to the Treehouse to celebrate. While at dinner, they get a call from Kayvon who says, "Why are you out celebrating? I'm here by myself trying to figure out why your final program still doesn't get peak performance." What is the problem that is limiting performance, and what percentage of peak is Kayvon observing?

Problem 3: Data-Parallel Thinking (45 points)

You are given a massively parallel machine with N processors (yes, one per input element) and asked by a colleague to produce a parallel algorithm for computing the average of the numbers in each of four sub-sequences in a sequence of numbers. For example consider a length-N sequence of numbers `input` and an array `seq_starts` indicating the start of each sub-sequence, as shown below:

```
input      = {3, 4, 6, 10, 20, 5, 1, 10, 3, 8}
seq_starts = {1, 0, 1, 0,  0,  1, 0, 1,  0, 0}
```

The result of computing the sum of all numbers in each of the three sub-sequences is a length-4 array below.

```
seq_avg = {3.5, 12, 3, 7}
```

To help you out, your colleague hands you a library with a highly optimized prefix sum and sort routines. (Recall prefix sum is an implementation of inclusive scan with the "+" operator.)

```
void sort(int count, int* input, int* output);
void prefix_sum(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 the methods provided (and its valid if you choose to not use all of them) and bulk launch of any function you wish to create, implement a data-parallel version of the sub-sequence averaging problem 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 id of the current invocation.

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


int   input[N];       // array of numbers, assume this is initialized
int   seq_starts[N];  // at most 4 of these elements will be 1
                      // you can assume seq_starts[0] = 1

float seq_avg[4];     // your result should go here
```