Full Name:_____

Andrew ID:_____

15-418/618, Practice

Exam 1

October 1, 2018 6pm

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The practice exam has a maximum score of **88** points. They will be graded for pass, pass-minus, fail / blank.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- On the actual exam, you may use 1 post-it of notes and 80 minutes. For practice, you should try to follow these guidelines.
- Submit your answers to Gradescope.

Do not write below this line

Problem	Your Score	Possible Points
1		5
2		5
3		12
4		26
5		14
6		17
7		9
Total		88

Data Parallel Programming with ISPC

Problem 1. (5 points):

Consider the following two implementations of a statically-scheduled ISPC routine that produces an `N`-element `output` array by performing a simple computation on an `input` array. Note that the only difference between these two versions is the scheduling: the real computation (i.e. doubling each input value and storing it as output) is the same. Also recall that in ISPC, *programCount* is the number of simultaneously executing instances in the gang, and *programIndex* is the ID of the current instance in the gang.

Version A:

```
export void doubler(uniform int N, uniform float* input, uniform float* output) {
   // assume N % programCount = 0
   uniform int count = N / programCount;
   int start = programIndex * count;
   for (uniform int i=0; i<count; i++) {
      int idx = start + i;
      float value = input[idx]*2.0f;
      output[idx] = value;
   }
}
```

Version B:

```
export void doubler(uniform int N, uniform float* input, uniform float* output) {
   // assume N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount) {
      int idx = i + programIndex;
      float value = input[idx]*2.0f;
      output[idx] = value;
   }
}
```

A. One of these two versions of the code is likely to be much faster than the other one, given that we are using the ISPC compiler. Explain which version is faster (`A` or `B`), and *why* that is the case.

Problem 2. (5 points):

Assume a GPU with 4 cores running at 1GHz, each able to queue 64 warps of 32 threads each. Every cycle the core will fetch and execute 1 instruction. (Keep in mind this instruction is executed on an entire warp in that cycle.) The main workload (shown below) is run with a 1M-element input array, `X[]` (initialized randomly to values between `0.0f` and `1.0f`) and with an output array `Y[]` (also with 1M elements). Assume that the input and output arrays are resident in CUDA device memory (i.e. the GPUs memory) at the time of the kernel launch.

```
__global__ void foo(float* X, float* Y) {
   // get array index from CUDA block/thread id
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   float result = 0.f;
   float val = X[idx]; // memory load

   if (val > 0.5f) // 2 arithmetic cycles
      result = doA(val); // 200 arithmetic cycles
   else
      result = doB(val); // 200 arithmetic cycles

   Y[idx] = result; // memory store
}
```

A. The performance of the workload is roughly *half* of the maximum, even after trying different sized thread blocks. Given your knowledge from 418, this should not be surprising. Please explain why this observation makes sense (given how GPUs would process the code above).

Comparing Programming Models: Shared Address Space vs. Message Passing

Problem 3. (12 points):

Recall that in class we used a simplifed Grid Solver kernel to highlight some of the differences between the *Shared Address Space* and *Message Passing* programming models. Given what we observed in that example code, *please contrast these two programming models* for each of the following aspects of orchestrating the parallel computation:

A. How data is *structured and accessed*.

B. How *communication* takes place.

C. How *synchronization* takes place.

D. How *replication* is managed.

Problem 4. (26 points):

Once we have decomposed a program into parallel tasks, we must decide how to assign these tasks to processors. There are both *static* and *dynamic* approaches for accomplishing this.

A. **Benefits of Static Scheduling**

(a) Describe an application scenario (from a source code perspective) where you would expect *static scheduling* to outperform dynamic scheduling.

(b) In this scenario, what aspect of *static scheduling* would enable it to outperform dynamic scheduling?

B. **Benefits of Dynamic Scheduling**

(a) Describe an application scenario (again, from a source code perspective) where you would expect *dynamic scheduling* to outperform static scheduling.

(b) In this scenario, what aspect of *dynamic scheduling* would enable it to outperform static scheduling?

C. **Tradeoffs with Dynamic Scheduling**

Even if one chooses to do purely dynamic scheduling (e.g., using distributed task queues), there is still a choice to be made regarding how much computation should be bundled together into a single task. For example, in the Wandering Salesman Problem (WSP), a single task in the task queue might be as small as considering one additional city along a given path, or it might be as large as considering an entire (large) subtree within the search space. Please answer the following questions regarding tradeoffs when choosing dynamic task sizes.

(a) Explain exactly why it is that choosing a dynamic task size that is *too small* can result in poor performance.

(b) Explain exactly why it is that choosing a dynamic task size that is *too large* can result in poor performance.

(c) Assume that when we run a dynamically-scheduled parallel program on a given shared-memory multiprocessor, the hardware can break down absolute execution time (aggregated across all processors) into the following three categories: (i) time spent *executing instructions*, (ii) time spent *stalled accessing data* (both local and remote), and (iii) time spent *stalled waiting for synchronization*. Based upon this information alone, as we vary the dynamic task size, describe what effects on these specific components of execution time would lead you to believe that:

i. the dynamic task size is *too small*?

ii. the dynamic task size is *too large*?

(d) **Task Queues**

What are the advantages and disadvantages of using distributed task queues (as opposed to a global task queue) to implement load balancing?

Problem 5. (14 points):

A. **Block vs. Row Decomposition.** Recall that during class, we spent time discussing the SOLVER kernel, which is a simplified version of the OCEAN application where each element `A[i][j]` is computed based upon its immediate neighbors (i.e. `A[i][j-1]`, `A[i][j+1]`, `A[i-1][j]`, and `A[i+1][j]`). Two options that we discussed for partitioning the data across the processors were:

Block Decomposition: each processor gets a contiguous *square chunk* of the matrix; and

Row Decomposition: each processor gets a contiguous *set of rows* (given row-major array accesses).

Assuming a shared-memory machine with invalidation-based cache coherence and $N >> P$, please answer the following questions.

(a) If the cache block size was a single word, would you expect *block* or *row* decomposition to perform better? Please explain your answer.

(b) As the cache block size grows larger, how would you expect this to affect the performance of each approach? Please explain your answers.

i. Impact of larger cache block size on *block decomposition*:

ii. Impact of larger cache block size on *row decomposition*:

B. You are writing a kernel, and trying to make it perform well on a parallel system. Using some analysis tools, you find that much of the time is being spent on memory accesses to the following data structure. You annotate the highly accessed fields with the access types.

```
struct process {                  // 64 bytes, fits in cache line
   int num_children;              // READ and WRITTEN by ALL processors
   signal_func* sigchld_handler; // READ by ALL processors (but not written)
   signal_mask* block_mask;       // READ by ALL, and WRITTEN by JUST ONE processor
   ...
};
```

Using your 418 knowledge, what is the problem and how might you fix it?

Cache Coherence

Problem 6. (17 points):

The following questions are related to cache coherence.

A. For a directory-based implementation of invalidation cache coherence, why is it that objects that are both frequently read and written (e.g., a shared task queue) are not expected to cause large numbers of invalidations upon writes?

B. Consider a machine with 32 byte cache lines that uses a flat, memory-based directory cache coherence protocol. For each of the two machine sizes below, compute the storage overhead of the "presence" state only (i.e. you can ignore the dirty and overflow bits) as a *fraction of main memory size* for the following two directory schemes: (i) full bit vector, and (ii) a limited pointer scheme with 3 pointers.

(a) 32 processors

(b) 1024 processors

C. Recall from lecture that some Intel and AMD processors supplement the **MESI** protocol with a *fifth* coherence state: either *"Forward"* (**F**) in the case of Intel, or *"Owned, but not exclusive"* (**O**) in the case of AMD. For a given cache line, *one* processor might be in one of these special states (**F** or **O**) while the other processors that have copies of that shared line are in the **S** state.

(a) While some details of the **F** state vs. the **O** state are different, they share a common performance benefit (relative to a plain **MESI** protocol). What is that benefit?

(b) Compared with the **O** state, an interesting feature of Intel's **F** state is that the processor that initiates a BusRd request enters the **F** state upon completion of the BusRd, and the processor that previously had the line in the **F** state is downgraded to an **S** state. (In contrast, the **O** state does not migrate between caches upon a BusRd.) What is the potential performance benefit of *migrating* the **F** state between caches (compared with not migrating it, in the case of the **O** state)?

Snoop-Based Multiprocessor Design

Problem 7. (9 points):

The following questions are related to the design of bus-based shared-memory multiprocessors. While keeping your answers brief, please be sure to explain the reasoning behind your answers.

A. Assume that we have an *atomic bus* and are following the MESI (i.e. *Modified-Exclusive-Shared-Invalid*) protocol. Processor *P1* has cache block `A` in a shared state. Processor *P1* executes an instruction that would like to write to a location in block `A`, and *P1's* cache controller requests the bus. While *P1* is still waiting to be granted the bus, another processor (*P2*) successfully acquires the bus, and issues a *read exclusive* for block `A` on the bus.

How should *P1's* cache controller react in order to maintain *sequential consistency*? Why?

B. When we initially described snoopy cache coherence protocols in this course, we assumed an *atomic bus*. (This was also the assumption in the question above.) Real bus-based multiprocessors, however, are built using a *split transaction bus*. Briefly describe one unique challenge that arises from having a split-transaction bus as opposed to an atomic bus, and describe a mechanism that can be used to overcome that challenge. (Hint: your answer to part A is not a valid answer to this question, since that question assumed an atomic bus.)