CMU 15-418/618 Spring 2015 Exam 1 SOLUTION

Warm Up: Miscellaneous Short Problems

Problem 1. (19 points):

A. (3 pts) Imagine you are asked to implement ISPC, and your system must run a program that launches 1000 ISPC tasks. Give one reason why it is very likely more efficient to use a fixed-size pool of worker threads rather than create a pthread per task. Also specify how many pthreads you'd use in your worker pool when running on a quad-core, hyper-threaded Intel processor. Why?

Using a fixed size pool of worker threads (that is created at program start or upon first launch) has the benefit of not introducing thread startup overhead for every bulk task launch. More importantly, by sizing the number of worker threads to the execution capability of the machine, the implementation minimizes the possibility of thrashing due to the working set of all threads exceeding important levels of the memory hierarchy of the machine.

It makes sense to create a pool of eight worker threads for this machine.

B. (3 pts) A key idea in this course is the difference between *abstraction* and *implementation*. Consider two abstractions we've studied: ISPC's `foreach` and the `pragma omp parallel for` construct. Briefly describe how these two abstractions have similar semantics (Hint: what do the constructs declare about the associated loop iterations? Careful, be very precise!). Then briefly describe how their implementations are radically different (Hint: consider their mapping to modern CPUs). As a reminder I'll give you two syntax examples below:

```
ISPC foreach:
foreach (i = 0 ... 100) {
  x[i] = y[i];
}

OpenMP:
#pragma omp parallel for
for (int i=0; i<100; i++) {
  x[i] = y[i];
}
```

In both cases, the constructs declare that the loop iterations are independent and can be executed in any order without violating program correctness. One of those valid orderings is a parallel ordering, and both ISPC and OMP use during mechanisms to implement parallel execution of the loops. ISPC uses vector instructions and most OMP implementations will use a pool of worker threads.

C. (3 pts) CUDA programmers are told variables stored in on-chip shared memory can be accessed with high-bandwidth and low latency (similar to how cached variables can be accessed efficiently on a CPU). Use an *abstraction vs. implementation* argument to describe why Prof. Kayvon says "CUDA shared memory is not a cache!"

*CUDA shared memory is a logical address space (abstraction) that is *implemented* using on-chip storage. Software, via loads and stores, moves between the global memory address space and the shared memory address space. In contrast, on the CPU, software is presented with a single logical address space. A cache is a hardware implementation detail that is hidden from software (yes, let's ignore modern cache hint instructions for now). Hardware chooses to locate some data in the cache to provide higher performance, but this decision is invisible to software and does not impact program correctness.*

D. (3 pts) Your friend suspects that his program is suffering from high communication overhead, so to overlap the sending of multiple messages, he tries to change his code to use asynchronous, non-blocking sends instead of synchronous, blocking sends. The result is this code (assume it's run by thread 1 in two-thread program).

```
float mydata[ARRAY_SIZE];
int dst_thread = 2;

update_data_1(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);

update_data_2(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);
```

Your friend runs to you to say "my program no longer gives the correct results." What is his bug?

The problem is that even though the first `async_send` *call returns, there is no guarantee that the send operation has completed at this time. As a result, the contents of the buffer* `mydata` *may be overwritten before the send completes. The receiver may receive incorrect data for the first message.*

E. (4 pts) Complete the ISPC code below to write an if-than-else statement that causes an 8-wide SIMD processor to run at nearly 1/8th its peak rate. (Assume the ISPC gang size is 8. Pseudocode for an answer is fine.)

```
void my_ispc_func() {

   int i = programIndex;


   if (i == 0) {

        very long sequence of instructions here!

   } else {

        very short sequence of instructions here!

   }
}
```

The code above will suffer from execution divergence. When running the 'if' clause, which occupies the bulk of execution since it involves a very long sequence of instructions, only one of the eight program instances will be doing useful work, so the system runs at 1/8 peak performance.

F. (3 pts) Assume you want to efficiently run a program with very high temporal locality. If you could only choose one, would you add a data cache to your processor or add a significant amount of hardware multi-threading? Why?

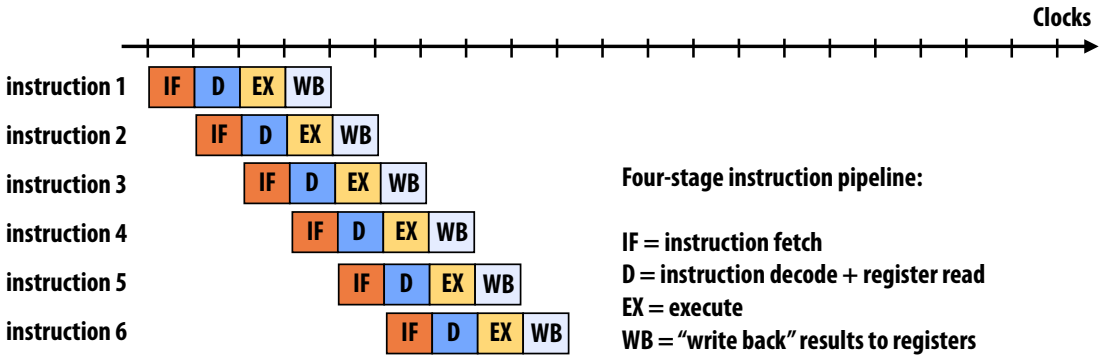
It makes sense to choose the cache. Because it exhibits high temporal locality, most data accesses in the program will be cache hits, allowing the processor to stay busy doing useful work. Hardware multi-threading is also a mechanism for keeping the processor busy by hiding latency (rather than eliminating it). However, to see benefits from multi-threading the program would need to be re-written to be parallel, and since no requests are absorbed by the cache, would need to have high memory bandwidth. As a result, the cache is a much simpler solution for this workload.

A Yinzer Processor Pipeline

Problem 2. (17 points):

Yinzer Processors builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. (1 pt) Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

1 instruction per clock. One instruction completes each cycle.

B. (1 pt) Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

1/4 instruction per clock. One instruction completes every four cycles.

C. (1 pt) What is the latency of completing an instruction?

4 cycles

D. (1 pt) Imagine the EX stage is modified to improve its throughput to two instructions per clock. What is the new overall maximum instruction throughput of the processor?

Throughput remains one instruction per clock. EX will only receive inputs at a rate of one instruction per clock, so its higher throughput ability goes unused.

E. (3 pts) Consider the following C program:

```
float A[100000];
float B[100000];
// assume A is initialized here

for (int i=0; i<100000; i++) {
    float x1 = A[i];
    float x2 = 2*x1;
    float x3 = 3 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop), what is the average instruction throughput of this program? (Hint: You should probably consider a number of loop iterations worth of work).

The throughput is 4/13 instructions per clock. All instructions within the loop body are dependent on each other, but the last instruction in the body is independent of the first instruction executed in the next iteration of the loop. Therefore there is a steady-state pattern of independent, dependent, dependent, dependent, ...

F. (6 pts) Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

The main idea is to "unroll the loop four times" and reorder instructions so that each dependent instructions are separated by four other instructions. Therefore, the prior instruction will have completed before the later instruction dependent on it issues.

```
for (int i=0; i<100000; i+=4) {
   float x1[4], x2[4], x3[4];

   x1[0] = A[i];    x1[1] = A[i+1];   x1[2] = A[i+2];   x1[3] = A[i+3];
   x2[0] = 2*x1[0]; x2[1] = 2*x1[1];  x2[2] = 2*x1[2];  x2[3] = 2*x1[3];
   x3[0] = 3+x2[0]; x3[1] = 3+x2[1];  x3[2] = 3+x2[2];  x3[3] = 3+x2[3];
   B[i] =  x3[0];   B[i+1] = x3[1];   B[i+2] = x3[2];   B[i+3] = x2[3];
}
```

G. (4 pts) Now assume the code is changed to the following:

```
#pragma omp parallel for
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this program, what feature would you add to the processor to obtain peak instruction throughput on a *single core*? Please be specific. (You may not change how the instruction pipeline works or how it handles dependent instructions. You may not change the program.)

Hardware multi-threading, with four-threads per core. OpenMP will implement the parallel for loop using a pool of worker threads. All instructions within a thread are still dependent, but the four-cycle latency of instruction execution can be covered by executing instructions from other threads. (instructions from different threads will be independent)

Running a CUDA Program on the GPU

Problem 3. (19 points):

The 15-418/618 TAs decide 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 (Prof. Kayvon 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. (2 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".)

8 cores × 16 operations/clock × 1 Ghz = 128 Gigaops/sec (128 operations per clock). In other words, there are 128 single-precision (SIMD) ALUs running at 1 Ghz.

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 thread-block size of 128 threads, and that enough thread blocks are created so there is one 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 (val % 2 == 0)                          // 2 arithmetic cycles
         result += doA(val);                     // 7 arithmetic cycles
      else
         result += doB(val);                     // 7 arithmetic cycles
   }

   Y[idx] = result;                              // memory store

}
```

B. (3 pts) The TAs hook the PKPU up to a memory system that provides **16 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[] = {1.0f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}` (correspondingly, the output array `Y[]` is 128 elements) and observe that it *does not* realize peak performance on the PKPU. Xiaofan is devastated. What is the primary reason for the low performance? (Please assume the input and output arrays are resident in CUDA device memory (in GPU memory) at the time of the kernel launch.)

This workload is too small for the PKPU. This program creates a single thread block of work, which will be run on one of the eight PKPU cores. As a result, it runs at most at 1/8 of peak throughput.

C. (3 pts) Cary 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.)

Both workloads generate plenty of work for the PKPU's 8 cores: 1024 thread blocks and 1M thread blocks respectively. Therefore, while the second program takes longer to execute (it does more work), both programs execute with the same throughput.

D. (4 pts) Vivek looks at the timing of Cary's experiments on the largest dataset and says "Aw shucks Cary, this program is not achieving peak utilization of the processors execution units." What is the problem limiting performance? What percentage of peak PKPU throughput does Vivek observe? Remember:

- The processor runs a instruction on an entire warp worth of CUDA threads in a single cycle.
- The memory system provides 16 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.)

Each warp's load requests 16 lanes × 4 bytes = 64 bytes of data. Therefore, across the 8 cores 512 bytes are requested. In a situation with no divergence (which is true in this case) this load occurs every 9 cycles, meaning that to keep up, the memory system must provide 512/9 = 56 GB/sec of bandwidth. The program is bandwidth bound and achieves about 28% of peak.

Note that there is no divergence in this case because, based on the TA's input, all threads in the warp simultaneously work on values separated by 1000, so all threads in the warp take the same branch in the conditional.

However, since the divergence behavior was subtle we all gave full credit to students that answered the question assuming divergence was present (the question certainly led you to believe this was the case and should have been worded better). In this case, the load occurs every 16 cycles, and thus 32 GB/sec of bandwidth are required, and the PKPU achieves 25% of peak throughput. 50% percent of the time the PKPU is not issuing instructions (stalled on memory) and when the PKPU is executing instructions, only have the lanes in the work were active.

E. (4 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). Arjun also **rewrites the program to sort the input array so all even numbers are at the front of the array and all odd numbers are at the end.** Cary, Arjun, Vivek, and Xiaofan congratulate themselves on a job well done and head out to Rose Tea Cafe to celebrate. While at dinner, they get a call from Will who says, "Why are you out celebrating, I'm here by myself trying to figure out why Arjun's program still doesn't get peak performance." What is the problem that is limiting performance, and what percentage of peak is Will observing?

*The program does not achieve peak throughput because even thought there is now sufficient bandwidth, the PKPU cannot fully hide the 100-cycle memory latency of loads. When a warp stalls on a memory request, there are seven other warps can be run to hide the latency of this operation. Each of those warps may issue up to 9 arithmetic instructions, so 7×9=63 of the stall cycles are hidden. (The GPU runs at 63% of peak.) Although not exactly correct, we also accepted the answer of 8*9=72% of peak for full credit.*

F. (3 pts) When the rest of the TAs return to the office with their Bubble Teas, Will is nowhere to be found. They find a note saying "Problem solved. I made a small tweak to the design of the PKPU and now Arjun's modified program runs at peak performance now. Heading home to watch Game of Thrones..." What was Will's tweak? (Note: we are looking for a change to the capabilities of the processor. The memory system is unchanged.)

Will modified the PKPU to have more warp execution contexts, and thus more latency hiding ability. Full credit was also given to answers such as adding support for prefetching.

Memory Consistency

Problem 4. (10 points):

A. (5 pts) Consider the following program which has four threads of execution. In the figure below, the assignment to `x` and `y` should be considered stores to those memory addresses. Assignment to `r0` and `r1` are loads from memory into local processor registers. (The print statement does not involve a memory operation.)

Processor 0	Processor 1	Processor 2	Processor 3

`x = 1`	`y = 1`	`r0 = y`
		`r1 = x`
		`print (r0 & ~r1)`
			`r0 = x`
			`r1 = y`
			`print (r0 & ~r1)`

- Assume the contents of addresses `x` and `y` start out as 0.
- Hint: the expression `a & ~b` has the value 1 only when `a` is 1 and `b` is 0.

You run the program on a four-core system and observe that both processor 2 and processor 3 print the value 1. Is the system sequentially consistent? Explain why or why not?

No, it is not. If processor 2 prints the value 1, that means that Y=1 and X=0 (so the write to Y came before the write to X). However, if processor 3 also prints 1, it means that X=1 and Y=0. There is no way to put the memory operations on a timeline that is consistent with these operations, this the system is not sequentially consistent.

THIS PAGE JUST DEFINES test_and_set. If you're familiar with test_and_set, move to the next page.

Test-and-set is an atomic primitive that is supported as a single atomic instruction on many machines. For reference, the logic of test-and-set is given below. If the value at address `addr` is the same as `old_value`, then `new_value` is stored to the location and the function returns true. Else, it returns false. (Keep in mind this entire sequence is executed atomically in a test-and-set operation as a single instruction. We've only written it as code below for clarity.)

```
// this is just provided for reference
bool test_and_set(int* addr, int old_value, int new_value) {
   if (*addr == old_value) {
      *addr = new_value;
      return true;
   }
   return false;
}
```

Note that a test-and-set instruction will be treated as a write by the coherence protocol (in other words, it requires exclusive access).

However, assume that like a read, subsequent program instructions that depend on the results of the test-and-set cannot be issued until the test-and-set completes and is visible to all processors.

The following code contains a critical section and assume it is run over and over by two cores in a tight loop. Assume reads/writes to `lock` and variables `A` and `B` are memory operations. Operations on `r0` manipulate registers and *are not* memory operations.

```
int lock;  // global variable

while (!test_and_set(&lock, 0, 1));  // acquire lock

r0 = A;


if (r0 == 100) {   // if A gets to 100, count the event and reset


   B++;


   A = 0;


} else {


   A = r0 + 1;

}

fence();   /* THIS IS THE ONLY FENCE REQUIRED */

lock = 0;  // release lock
```

B. (5 pts) Assume the processor adopts a memory consistency model that relaxes **write-after-write ordering**. Please describe what undesired program behavior can occur this scenario (why?). Then insert the minimum number of memory fences necessary to ensure correct operations (just write "fence" in places in the code above). Justify your code.

We don't want the write that releases the lock to be observed by other processors before writes to values in the critical section, so there needs to be a fence prior to `lock=0`*. Without this fence the operations in the critical section are not carried out atomically.*

Rendering Particles

Problem 5. (18 points):

Consider the following code that produces a grayscale image from a large array of single pixel particles. (Pixel magnitudes are represented as an int.) The code computes the color of each particle and then accumulates that color into the final output image.

```
struct Particle {
  int x, y, density;
};

Lock l;
int image[512][512];      // 1MB
Particle particles[N];    // assume N is very large 100's of millions

// assume thread 1 executes splat on the 1st half of the array 'particles'
// assume thread 2 executes splat on the 2nd half of the array 'particles'

void splat(Particle* particles, int num) {
   for (int i=0; i<num; i++) {
      Particle* p = &particles[i];
      lock(l);
      image[p->x][p->y] += compute_color(p);
      unlock(l);
   }
}
```

Assume that:

- The code is run on a two-core processor with a single (shared) fully associative, 4 MB cache.
- The load of particle data is free (e.g., due to prefetching). **For simplicity assume only the image data is stored in the cache.**
- Acquiring a lock takes 10 cycles. Releasing the lock takes 10 cycles.
- `compute_color` only reads data from the input particle and requires 1000 cycles
- The store to the output image takes 1 cycle if the address is cached, or 75 cycles on a cache miss.
- **To keep things simple, assume the processor executes all instructions in order (it does not proceed to the next operations until the current one is finished) and there is infinite memory bandwidth**.

A. (3 pts) Assuming the position of the particles are randomly distributed throughout the image (uniform distribution), approximately what speedup (relative to a single threaded implementation without the lock) do you expect to observe running the code on the two core processor? Why? (a ballpark number is fine)

Essentially no speedup. The bulk of the computation is carried out in the critical section, and thus most of the program's execution is serialized.

B. (3 pts) Describe a simple fix to the code that will significantly improve its performance. Your fix should use no more than one lock. Approximately what speedup do you expect now? (again, a ballpark number is fine)

Move the call to `compute_color` *out of the critical section. The execution of this function need not be atomic, only the update to the final image. The result of this change is that the program should realize a 2× speedup.*

C. (6 pts) Now assume the implementation of `compute_color` is changed to only take 50 cycles. Describe how you might modify the code to again significantly improve its performance. Please write pseudocode (although it need not be compilable C code). **You may modify the code however you wish, but you are allowed to use no more than one lock.**

Now the 20 cycles of overhead incurred by taking the lock are quite significant. One solution is to split the processing of particles across processors, and avoid the need for a lock by having each thread update its own private copy of the image. Then, at the end of the computation, merge the two results. Since the processing of the subimages dominates execution time, the program will achieve near 2× speedup at the cost of extra space overhead.

An alternative solution divided the image in two, and gave responsibility for processing each half of the image to each thread. Both threads iterate over all particles, and only process particles whose position falls in the region of the image they were responsible for.

A number of solutions were on the right track, but provided incorrect loop indexing that gave incorrect results.

D. (3 pts) Now assume the cache is shrunk to 1 MB in size. Does your implementation in part C perform better or worse than your version of the code with locks? Why? (In this problem, assume `compute_color` still takes 50 cycles.)

For solutions that duplicated the image data, this solution would cause thrashing of the cache (the working set is now 2MB). The resulting cost of memory stalls would result in lower performance than the implementation with locks.

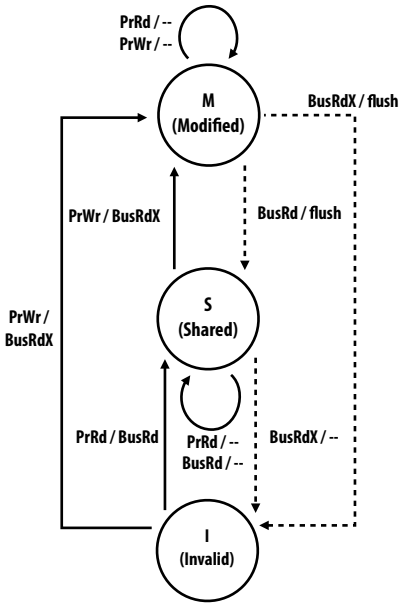
Other student solutions to the parallelism problem were not affected by the change in cache size, so we graded accordingly on a case-by-case basis.

E. (3 pts) If you were told you could assume that all pixels in the output image could be clamped to the value 255 would your answer to part D change? (Please describe any modifications to the code you would make under this new assumption.)

You could store image pixels as chars, rather than ints. This modification would reduce the size of the working set by 4, making it fit in cache.

Cache Coherence

Problem 6. (17 points):



Your friend suggests modifying the MSI coherence protocol (shown above) so that the PrWr/BusRdX behavior on the I-to-M and S-to-M transitions is changed to PrWr/BusRd.

A. (4 pts) Describe how this modification does not ensure the **write propagation requirement** of coherent memory systems. (A good answer will describe a sequence of memory operations that demonstrates the problem.)

Consider the case where P1 seeks to modify data where the line is in the S state in P1. Since only a BusRd goes out on the bus, the line will be in the M state in P1, but remain in the S state in P2. If the modified line in P1 is never flushed, the write never propagates to P2.

B. (4 pts) In class I mentioned that modern multi-core Intel processors use a directory-based protocol to keep per-core L2 cache's coherent. The processors have a shared L3 cache, and **the directory is stored entirely** in the L3 cache (not in main memory). Briefly, describe why the directory scheme can avoid having an entry for each line in main memory (hint: inclusion).

The only lines whose coherence status needs to be tracked are the lines in the L2 caches. Due to inclusion, any line in the L2 cache is also in the L3, so the directory information for the line can be stored with the line in the L3.

C. (3 pts) Prof. Kayvon decides to take a road trip to Seattle. He wants to take a quick route, so he writes a program that generates many paths from Pittsburgh to Seattle through 10 random cities. The code, given below, computes the length of the shortest of these paths.

```
int best_so_far = HUGE_NUMBER;   // global variable
int counter = 0;                 // global variable

#pragma omp parallel for schedule dynamic  // dynamic scheduling
for (int i=0; i<NUM_PATHS; i++) {
   length = compute_random_path_length();  // compute a random path
   if (length < best_so_far) {
      // atomically perform: best_so_far = min(length, best_so_far)
      // (performs a **write** to address of best_so_far)
      atomic_min(length, &best_so_far);
   }
}
```

Assuming paths have random length, notice that the program will frequently find "best paths" early in its execution (when `best_so_far` is large) but very rarely find new best paths late in the program since the bound has grown tight.

Consider execution during the earlier phases of execution on a snooping-based multi-core processor. Do you think an invalidation-based or update-based coherence protocol implementation would be more efficient solution for this program. Why?

Early in program execution, there are many writes to a shared variable that is constantly read. An update-based protocol would immediately send those changes to all other caches. As a result, reads by the other processors will not trigger caches misses like they would in an invalidation-based protocol.

D. (3 pts) Xiaofan learns about Prof. Kayvon's plans and wants to tag along. Trying to help, he says "I think that a directory-based coherence protocol would be preferable to a snooping-based implementation for the later stages of your program. Prof. Kayvon retorts, "First, I think snooping is a bit better... also, it doesn't matter anyway." Why does Prof. Kayvon disagree, and why did he dismiss the importance of the debate?

Since updates are very infrequent during the later stages of computation, it is likely that many caches will have the line in the shared state. As a result it is likely more efficient to broadcast the BusRdX to all caches, rather than send point to point messages to a large number of shares. However, since writes are infrequent, the performance of the writes is likely not that important to overall program performance, so that's why Prof. Kayvon dismisses the important of the debate.

E. (3 pts) Trying to atone for his mistake, Xiaofan tries to improve the performance of the code. He removes the if check prior to the `atomic_min`, noticing that it's redundant with the check that the `atomic_min` performs. He's aghast when the performance of the program drops. Why did this happen? (Hint: relative to the original program, the modification is particularly bad during late-stage execution of the program.)

Recall the `atomic_min` operation is a write (even if the new path length doesn't change the value of `best_so_far`, the write of the result of the min operation will always occur). By removing the if check, Xiaofan's change causes this write to occur each iteration of the loop. Before this change, the write occurred only if the new path length was less than `best_so_far`. Not every iteration of the loop takes a cache miss on `best_so_far`.