ECE 459 W18 Final Exam Solutions

P.Lam, J. Zarnett

March 30, 2018

(1)

(a) *Local memory* is shared between work-items belonging to the same work-group; it is like a user-managed cache.

Constant memory resides on the GPU, and is cached. It cannot be changed.

(b) The problem is the lock here has been incorrectly split: thread 1 wants to insert a node with `id` of 9; it gets the lock, checks, finds that item is not in the list, and then unlocks the lock. At this point there is a thread switch to Thread 2 which also wants to insert `id` 9: it finds that it is not in the list, and inserts the node with 9 into the list. Then thread 1 resumes and it also puts 9 in the list, so now the list has two entries with 9.

The solution is to remove the unlock and lock statements in the middle of the function (immediately before the `n->next = l->head)` statement.

(c) The easiest way is to compare is make a table, since things are discrete here with the GPU.

Time (s)	CPU units	GPU units
0	0	0
0.12	120	0
0.22	220	200
0.32	320	400

We can see that after 0.32s, the GPU can complete 400 units, while the CPU would only have completed 320.

It's also OK if you assume that the GPU computes units linearly once you've paid startup cost. In that case you could use linear equations and you'd get a breakeven point for the GPU of 240 units, at $t = 0.24s$.

(d) A lot of space is wasted by keeping a boolean value (1 bit, really) 4 bytes. Given a sufficiently large array this means the array will be spread across more pages of virtual memory and this will increase the number of cache misses. The solution is to pack your structures; you can use a smaller type (eg `char`) or bitfields to represent the boolean values. Check that your structures end up not aligning the chars on 4-byte boundaries.

(e) Missing points can be filled in with interpolation (i.e. average some adjacent values, for example). You need to know that your interpolation respects the underlying distribution of the function for interpolation to be valid.

(f) We have violated the `restrict` keyword's rules by having the two pointers alias. This could lead to incorrect output or other undesirable behaviour because we have violated the contract of `restrict`. The compiler is entitled to compile the code however it wants, and is not required to produce valid code. In particular, it might reorder `src` and `dest` in a way that breaks the code. (But it's allowed to do worse. It is allowed to generate literally any code.)

(g) video compression pipeline: (1) read input; (2) comprses; (3) write output. Compress could use multiple threads.

(h) Yes: it needs to be irreducible, positive recurrent, and aperiodic. Irreducible means that it's possible to get from one state to any other, and if state is the number of jobs in the queues then this is certainly possible; positive recurrence means we revisit states given long enough time (e.g., all queues are empty) so we can consider that met; aperiodicity is harder to prove, but given sufficiently random generation of random numbers then it will also be aperiodic.

(i) Make sure the cost of instrumentation is less than the cost of the execution; run with sufficiently large inputs both instrumented and uninstrumented, and compare running times.

(j) $\rho = 0.0277\ldots, s = 0.1s \Rightarrow \lambda = \frac{1000}{3600} = 0.27$ Hz, $T_q \approx 0.10s$.

(2)

```
question 2:

widget_result * load_and_process(int id) {
  pthread_t process_thread;
  widget * w = (widget *) map_get(widget_map, id);
  pthread_create(&process_thread, NULL, process_func, w);
  widget * ww = load_by_id(id);
  widget_result * r;
  pthread_join(process_thread, &r);
  if (ww->lastupdate != w->lastupdate) {
    map_put(widget_map, id, ww);
    r = process(ww);
  }
  return r;
}

void process_func(void * p) {
  widget * w = (widget *)p;
  widget_result * r = process(w);
  pthread_exit(r);
}
```

(3.1)

1. Wrong variable scoping on `i` (set to private, not shared)

2. The `grand_sum` variable has a race condition (fix by declaring it as a reduction on + or through atomic/critical)

3. Memory allocation is in the parallel section (should be outside)

(3.2)

```
pthread_mutex_t lock;

int main( int argc, char** argv ) {

 pthread_mutex_init( &lock, NULL );
 struct job* jobs = prep_tasks();
 pthread_t threads[NUM_TASKS];

 for( int i = 0; i < NUM_TASKS; ++i ) {
    pthread_create( &threads[i], NULL, work, &jobs[i] );
 }

 for( int i = 0; i < NUM_TASKS; ++i ) {
    pthread_join( threads[i], NULL );
 }

 free( jobs );
 pthread_mutex_destroy( &lock );
 return 0;
}

void* work( void* arg ) {
  struct job* j = (struct job*) arg;
  do_work( *j );
  pthread_mutex_lock( &lock );
  completed++;
  pthread_mutex_unlock( &lock );
  pthread_exit( NULL );
}
```

Other notes: yes, the question says “untieduns” instead of “untied”... Just ignore that...

Use of atomic integers or similar is okay, as is using the static initializer macro for the lock. Leaving off the pthread_exit call at the end of work is fine, as are other ways of passing the array around as long as you’re not taking the address of something on the heap.

Remember to clean up things you allocate.

(4)

```
int length;
cl_uint sums = malloc( iterations * sizeof( cl_uint ) );

for ( length = iterations; length > CL_DEVICE_MAX_WORKGROUP_SIZE; length = length /
    CL_DEVICE_MAX_WORKGROUP_SIZE ) {

  int new_length = (length / CL_MAX_WORKGROUP_SIZE);
  cl::Buffer in = cl::Buffer( context, CL_MEM_READ_ONLY, length * sizeof(cl_uint));
  cl::Buffer out = cl::Buffer( context, CL_MEM_WRITE_ONLY, new_length * sizeof(cl_uint));
  queue.enqueueWriteBuffer(in, CL_TRUE, 0, length * sizeof(cl_uint), sums);

  sumGPU.setArgs(0, in);
  sumGPU.setArgs(1, out);

  queue.enqueueNDRangeKernel(sumGPU, cl::NullRange, cl::NDRange(length), cl::NDRange(
      CL_DEVICE_MAX_WORKGROUP_SIZE ));

  queue.enqueueReadBuffer( out, CL_TRUE, new_length * sizeof(cl_uint), sums);
}

cl_int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < length; i++) {
  sum += sums[i];
}
float pi = (float) sum / iterations * 4;

free ( sums );
```

(5) 1. Constant propagation (various places, e.g. `C = 2` in `TT` to its use).

- requires no writes to `C` (one way to know for sure: never take address of `C`).
- replace `C` by 2.
- no need to load `C`'s value from memory.

2. Dead code elimination (def'n and use of `prod`).

- assuming `bar` has no side-effects (which is true here), and that `prod` is never used, can eliminate the assignment to `prod` and its declaration.
- faster because doing less work.

3. Devirtualization (virtual call to `t.foo()`).

- need to know `t` is always a `TT`.
- only instantiation of a `T` is of a `TT` in the whole program here.
- replace virtual call by nonvirtual call to `TT.foo`.
- branch prediction of nonvirtual calls potentially faster.

4. Inlining (call to `bar()`)

- for `bar`, no requirements for validity (though some for performance), so is always allowed.
- put body of `bar` into `function_to_optimize` to optimize.
- eliminates call overhead.

5. Common subexpression elimination (`i % 2`).

- if `i` is never written-to between the first and second uses.
- can compute `i % 2` once and store in a temporary variable:

```
t = i % 2;
if (t != 0) {
  sum += t;
}
```

- faster because it does less work.

(6.1) Strengths: good at identifying call graph edges; also good at finding which functions run for a long time via sampling.

Weakness: interpolates values so not very good at telling you the contribution of children functions to parents.

(6.2)

Some reasonable sample answers:

1. Log runtimes and the value of `mode` so we can determine whether `mode == 0` is the slow case, `mode != 0` is the slow case, or both. This will help to narrow down where the problem lies (if it does narrow it down).

2. Code inspection: open up the code and look through the code for any calls that we know are likely to be slow, e.g., malloc() or I/O.

3. Use a different tool (e.g., cachegrind) to learn more about why the operations are slow: if we have a lot of cache misses, for example, cachegrind will find them which gives a suggestion of where our time is going.

Also doable: discuss with co-worker what else they have discovered (why repeat work?)

(6.3)

Some reasonable sample answers:

1. Unnecessary computation – can we re-use a stored/computed or cached value (see speculation question)?

2. Poor choice of algorithm or data structure – would it be better to use a different algorithm given the access patterns for our data? e.g., use a more efficient sort algorithm, or change to a tree from a linked list.

3. Lock/resource contention – attempt to rewrite the code so that it does not need to lock shared data (e.g., use atomics) or to have its own copy of shared data.