Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Final
Winter 2018
ECE 459

Times: Tuesday 2018-04-10 at 09:00 to 11:30
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 3706311
Sections: ECE 459 LEC 001,002
Instructors: Jeff Zarnett, Patrick Lam

Open Book

Candidates may bring any reasonable aids. Open book, open notes. Calculators without communication capability permitted.

Instructions:

1. This exam is open book, open notes, calculators with no communication capability permitted.
2. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be interpreted as an academic offence.
3. There are six (6) questions. Not all are equally difficult.
4. The exam lasts **150** minutes and there are 100 marks.
5. Verify that your name and student ID number is on the cover page.
6. If you feel like you need to ask a question, know that the most likely answer is "Read the Question". No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
7. Do not fail this city.
8. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Marking Scheme (For Examiner Use Only):

Question	Mark	Weight
1		25
2		15
3		15
4		15
5		15
6		15
Total		**100**

Please initial:

1 Short Answer [25 marks total]

Answer these questions using at most three sentences. Each question is worth 2.5 points.

(a) OpenCL offers 4 types of memory: global, local, constant, and private. Explain what local and constant are for.

(b) The following function has a concurrency problem. Explain what the problem is (including an example of an unwanted behaviour) and how to fix it.

```
void insert_if_not_exists( struct list * l, struct node * n) {

  pthread_mutex_lock( l->lock );
  struct node* p = l->head;
  while( p != 0 ) {
    if ( p->id == n->id ) {
      pthread_mutex_unlock( l->lock );
      return;
    }
    p = p->next;
  }
  pthread_mutex_unlock( l->lock );

  pthread_mutex_lock( l->lock );
  n->next = l->head;
  l->head = n;
  pthread_mutex_unlock( l->lock );
}
```

(c) Assume that it takes 0.12s between starting a no-op OpenCL kernel and waiting for the results, and assume that you have a problem that the GPU can compute at 100% efficiency. The CPU can compute 1000 units per second[1]. The GPU can compute 200 work items simultaneously, and can therefore complete between 1 and 200 work items every 0.1 seconds. How many units will it take for the GPU to be faster than the GPU?

(d) You have discovered that `sizeof(bool)` on some platforms (including PowerPC) is 4 bytes (like an `int`). Suppose you have an especially large array of some `struct` that includes boolean attributes. Performance is a concern. What change would you make to your structure and why?

[1] Yes, the orders-of-magnitude are not reflective of today's computers, but they are easy for you to compute with.

Please initial:

(e) You are doing parallel execution of a mathematical function that has certain points that take a long time to evaluate. Your program terminates slow running threads "early", leaving some points missing. Explain how you could fill in those missing points (and what assumptions are required for this to make sense).

(f) Suppose a function `int send_recv( char* restrict src, char* restrict dest )` is invoked with the arguments `send_rcv( buffer1, buffer1 )` . What could go wrong here, and why?

(g) Describe an application (e.g. a computer program/system) that would combine the parallelization patterns "pipeline of tasks" and "single task, multiple threads".

(h) Is the program in assignment 4 a simulation of an *ergodic* system? Provide intuition as to why or why not.

(i) One problem with any program instrumentation is that it changes the observable behaviour of the program. This can happen when you add instrumentation for profiling. Let's say that you've instrumented your system to collect data (as you should). How can you ensure that the data that you collect are valid?

(j) Assume a M/M/1 open system. Requests come in at the rate of 1,000 per hour. The server takes 0.1s to respond to a request. What is the utilization ρ and the average completion time?

Please initial:

2 Speculation [15 marks]

Suppose that data elements in your program are of type `widget` shown partially below.

```
typedef struct {
  int id; /* Globally unique ID for a widget */
  long lastupdate; /* Last update timestamp */
  /* Various other properties that are not relevant to the question */
} widget;
```

You will add speculative execution of widget processing by modifying the `load_and_process` function.

```
widget_result* load_and_process( int id ) {
  widget* w = load_by_id( id );
  widget_result* r = process( w );
  return r;
}
```

Both the call to `load_by_id()` and `process()` take a long time. To do this speculatively then, you will introduce a cache. While the desired widget is loading, we can (in parallel) process the cached version of the widget, if that widget is found in the cache. If the `lastupdate` value of the loaded widget is not the same as the cached version (or the widget is not found in the cache at all), then put the loaded version in the cache and process the loaded widget. If the `lastupdate` value of the loaded widget matches the one in the cache, the speculatively-executed result is valid.

This system has a Map-like structure called `Map` to use for the cache. You use it with the following functions:

```
void map_put( Map map, int key, void* value );
void* map_get( Map map, int key ); /* Returns NULL if no matching value found */
```

Assume also the existence of a global variable `widget_map` of type `Map` that has been properly initialized. Use pthreads to introduce speculative execution to this function as described above.

3 OpenMP [15 marks total]

3.1 Program Analysis [6 marks]

Consider the following OpenMP-annotated C program fragment. Find 3 problems with the code and explain how to fix them (assume function `record_grand_sum` is defined elsewhere):

```
1  double* multiply_matrices( double* a, double* b, int rows, int cols) {
2    double* result;
3    int i, j, k;
4    double grand_sum;
5    grand_sum = 0;
6    #pragma omp parallel shared(result, a, b, i, grand_sum) private(j, k)
7    {
8      result = malloc( rowsA * colsB * sizeof( double ) );
9
10     #pragma omp for
11     for ( i = 0; i < rows; i++ ) {
12       for ( j = 0; j < cols; j++ ) {
13         for ( k = 0; k < cols; k++ ) {
14           result[i][j] += a[i][k] * b[k][j];
15           grand_sum += result[i][j];
16         }
17       }
18     }
19   }
20   record_grand_sum( grand_sum );
21   return result;
22 }
```

3.2 No OpenMP, No Problem [9 marks]

Recall this OpenMP-annotated program that is (almost) the solution to a midterm exam question. But now you are faced with a system where you cannot use OpenMP. Assume that `completed` is used elsewhere in a UI thread. Convert this program to its `pthread` equivalent.

```
int main( int argc, char** argv ) {
  /* Do other work */
  #pragma omp parallel
  {
    #pragma omp master
    {
      struct job* jobs = prep_tasks();

      for( int i = 0; i < NUM_TASKS; ++i ) {
        #pragma omp task untieduns
        {
            do_work( jobs[i] );
            #pragma omp atomic
            completed++;
        }
      }
      #pragma omp taskwait
      free( jobs );
    }
  }
  return 0;
}
```


Please initial:

4 $\sqrt{-1}\ 2^3\ \Sigma$ More OpenCL π [15 marks]

In the Winter 2017 final exam, a question focused on the computation of pi (π) using a Monte Carlo method. Random points are generated and we count the number of those points that are in the first quadrant of the circle, and then multiply that by 4 to get an estimate for pi. The way that this code worked, the OpenCL kernel generated a number of points, tested if they were in the first quadrant, and put either a 1 or 0 into an output array. Then there was the following code to produce the estimate of pi:

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

The for-loop will be slow in the CPU if the number of points is large. It could be replaced with an OpenCL kernel. The good news is the magical internet has provided you with a reduction kernel that does what you need:

```
// Original from https://dournac.org/info/gpu_sum_reduction
__kernel void sumGPU ( __global const uint *input, __global uint *partialSums, __local uint *localSums) {
  uint local_id = get_local_id(0);
  uint group_size = get_local_size(0);

  localSums[local_id] = input[get_global_id(0)];

  // Loop for computing localSums : divide WorkGroup into 2 parts
  for (uint stride = group_size / 2; stride > 0; stride /= 2) {
      // Waiting for each 2x2 addition into given workgroup
      barrier(CLK_LOCAL_MEM_FENCE);

      if (local_id < stride)
        localSums[local_id] += localSums[local_id + stride];
     }

  if (local_id == 0)
    partialSums[get_group_id(0)] = localSums[0];
 }
```

The maximum size of a workgroup on the given GPU is `CL_DEVICE_MAX_WORK_GROUP_SIZE`. Assume (for the sake of simplicity) the desired number of points is a multiple of that maximum size and everything divides evenly (therefore you do not need to worry that the last work group is not full or needs to be padded). Assume that all OpenCL setup is done and the first kernel has run and the output has been read out of the buffer into a CPU-side array of unsigned integers (`cl_uint`) called `results`. The initial size of the buffer should be `iterations`, the number of points generated, which is the same as the size of the `results` array. Here, the context is called `context`, the queue is `queue` and the kernel has been compiled as `sumGPU`.

Write the host code below to (1) create any buffers you will need, (2) enqueue those buffers, (3) set kernel arguments, (4) run the `sumGPU` kernel, and (5) read any results. The kernel should be run repeatedly, until the number of elements in the output of the kernel is smaller than `CL_DEVICE_MAX_WORK_GROUP_SIZE`. At that point, just use the CPU code (similar to the above) to sum the rest and produce the value for pi.

Please initial:

5 Compiler Optimization [15 marks]

Here's a C++ program (that is rather C-like, but still valid). Propose 5 different valid compiler optimizations for it: name the optimization, indicate what conditions are required for it to be valid, say what it would do, and explain why the resulting code could be faster (3 marks each).

```
#include <stdlib.h>
#include <stdio.h>

#define ROWS 5000
#define COLS 5000

int ** array;

class T {
public:
  virtual int foo(int x) {
    return x;
  }
};

class TT : public T {
  const int C = 2;
public:
  virtual int foo(int x) {
    return x * C + 1;
  }
};

int bar(int x) {
  int sum = 0;
  for (int i = 0; i < x; ++i) {
    if (i % 2 != 0) {
      sum += i % 2;
    }
  }
}

void function_to_optimize(T t) {
  int sum = 0, prod = 1;
  for (int i = 0; i < ROWS; ++i) {
    for (int j = 0; j < COLS; ++j) {
      sum += t.foo(array[bar(i*2)][j]);
      prod *= array[bar(i*2)][j];
    }
  }
  printf("%d\n", sum);
}

int main() {
  TT tt;
  array = (int **)malloc(sizeof(int *) * ROWS);
  for (int i = 0; i < ROWS; ++i) {
    array[i] = (int *)malloc(sizeof(int) * COLS);
    for (int j = 0; j < COLS; ++j) {
      array[i][j] = 5;
    }
  }
  function_to_optimize(tt);
  for (int i = 0; i < ROWS; ++i) {
    free(array[i]);
  }
  free(array);
  return 0;
}
```


Please initial:

6 Profiling [15 marks total]

Here's a function. You don't know very much about it, since I've omitted the details.

```
void suspicious_function( int mode ) {
  if (mode == 0) {
    // does something potentially slow
  } else {
    // does something else potentially slow
  }
}
```

Your coworker tells you that `suspicious_function` takes 40% of the entire program's runtime. You know that `suspicious_function` is called from functions `a()` and `b()`. Your task is to speed up the program.

6.1 `gprof` (3 marks)

Your coworker did not say which profiler came up with the 40% number. Let's say that it was `gprof`. Write down two strengths and one weakness of the `gprof` tool when applied to analyzing `suspicious_function`.

6.2 Next Steps (6 marks)

You need to investigate this situation further. Write down three steps you could take to better understand what's going on with `suspicious_function`, and explain what results these steps might provide you (if they yield any valuable results).

6.3 Fixing the problem (6 marks)

The description above left out any information about why `suspicious_function` might be slow. Speculate about three possible causes for its slowness and explain how you would address these causes.

Please initial:

OpenCL Reference

The following constructors may be helpful:

```
cl::Buffer (const Context &context, cl_mem_flags flags,::size_t size, void *host_ptr=NULL, cl_int *err=NULL)
cl::NDRange ()
cl::NDRange (::size_t size0)
cl::NDRange (::size_t size0,::size_t size1)
cl::NDRange (::size_t size0,::size_t size1,::size_t size2)
```

Kernel arguments can be set with set with:

```
void setArg( int argumentIndex, cl::Buffer buffer )
```

Note this is not actually the signature but you can use this and it works.

The following functions can be invoked on an instance of `cl::CommandQueue`:

```
cl_int  enqueueReadBuffer (const Buffer &buffer, cl_bool blocking,::size_t offset,::size_t size, void *ptr, const
     VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueWriteBuffer (const Buffer &buffer, cl_bool blocking,::size_t offset,::size_t size, const void *ptr
    , const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueCopyBuffer (const Buffer &src, const Buffer &dst,::size_t src_offset,::size_t dst_offset,::size_t
    size, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueReadImage (const Image &image, cl_bool blocking, const size_t< 3 > &origin, const size_t< 3 > &
    region,::size_t row_pitch,::size_t slice_pitch, void *ptr, const VECTOR_CLASS< Event > *events=NULL, Event *
    event=NULL) const

cl_int  enqueueWriteImage (const Image &image, cl_bool blocking, const size_t< 3 > &origin, const size_t< 3 > &
    region,::size_t row_pitch,::size_t slice_pitch, void *ptr, const VECTOR_CLASS< Event > *events=NULL, Event *
    event=NULL) const

cl_int  enqueueCopyImage (const Image &src, const Image &dst, const size_t< 3 > &src_origin, const size_t< 3 > &
    dst_origin, const size_t< 3 > &region, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueCopyImageToBuffer (const Image &src, const Buffer &dst, const size_t< 3 > &src_origin, const
    size_t< 3 > &region,::size_t dst_offset, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueCopyBufferToImage (const Buffer &src, const Image &dst,::size_t src_offset, const size_t< 3 > &
    dst_origin, const size_t< 3 > &region, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

void *  enqueueMapBuffer (const Buffer &buffer, cl_bool blocking, cl_map_flags flags,::size_t offset,::size_t
    size, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL, cl_int *err=NULL) const

void *  enqueueMapImage (const Image &buffer, cl_bool blocking, cl_map_flags flags, const size_t< 3 > &origin,
    const size_t< 3 > &region,::size_t *row_pitch,::size_t *slice_pitch, const VECTOR_CLASS< Event > *events=
    NULL, Event *event=NULL, cl_int *err=NULL) const

cl_int  enqueueUnmapMemObject (const Memory &memory, void *mapped_ptr, const VECTOR_CLASS< Event > *events=NULL,
    Event *event=NULL) const

cl_int  enqueueNDRangeKernel (const Kernel &kernel, const NDRange &offset, const NDRange &global, const NDRange &
    local, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueTask (const Kernel &kernel, const VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueNativeKernel (void(CL_CALLBACK *userFptr)(void *), std::pair< void *,::size_t > args, const
    VECTOR_CLASS< Memory > *mem_objects=NULL, const VECTOR_CLASS< const void * > *mem_locs=NULL, const
    VECTOR_CLASS< Event > *events=NULL, Event *event=NULL) const

cl_int  enqueueMarker (Event *event=NULL) const

cl_int  enqueueWaitForEvents (const VECTOR_CLASS< Event > &events) const

cl_int  enqueueAcquireGLObjects (const VECTOR_CLASS< Memory > *mem_objects=NULL, const VECTOR_CLASS< Event > *
    events=NULL, Event *event=NULL) const

cl_int  enqueueReleaseGLObjects (const VECTOR_CLASS< Memory > *mem_objects=NULL, const VECTOR_CLASS< Event > *
    events=NULL, Event *event=NULL) const

cl_int  enqueueBarrier () const

cl_int  flush () const

cl_int  finish () const
```


Please initial: