1. WARM-UPS – TRUE/FALSE (20%)

Indicate the correct answer with a clear "X" in the appropriate column.
It is NOT necessary to justify your answers on TRUE/FALSE questions, unless requested.

NOTE: You will get a -1% negative score for each wrong answer and 0 for not answering or answering both TRUE and FALSE.

No.	Question	TRUE	FALSE
a)	An efficient parallel implementation of a serial program is found by finding an efficient implementation of each step of the serial program		
b)	Separation of memory and CPU is often called the von Neumann bottleneck		
c)	Caching may provide superlinear speed-ups		
d)	The most common cache eviction policy is Least Recently Used		
e)	Flynn's taxonomy includes SIMD and MIMD		
f)	Multi-threading is considered a coarse-grained parallelism		
g)	Strongly scalable programs always improve performance with no. of cores/processors.		
h)	Weakly scalable programs follow Gustavson's Law		
i)	Elster's Bit-Reversal algorithm is O (N log N)		
j)	Snooping cache coherence takes advantage of bus architectures		
k)	One-sided communication is different from remote memory access		
l)	Amdahl's law says that if a fraction r of a serial program remains serial, then we cannot get a better speed-up than 1/r. This means we MUST resort to task parallelism in order to be able to scale further.		
m)	MPI_Broadcast does not need tags		
n)	MPI_Reduce may use the same buffer for both input and output		
o)	CUDA is less verbose than OpenCL, but OpenCL is offered on a wider selection of devices.		
p)	CUDA warps may be synchronized		
q)	CUDA uses sychthreads() to synchronize across SMs		
r)	The OpenCL-equivalent of a CUDA warp is called team		
s)	Coalescing memory on GPUs improves efficiency by using strided memory locations		
t)	Some recent CUDA devices give you the option to trade-off amount of shared memory with caching.		

2. PARALLEL COMPUTING BASICS (10%)

2a) What are the two main differences between threads and processes?

Threads: _____

Processes: _____

2b) Instruction Level Parallelism (ILP) attempts to improve processor performance by utilizing several functional units simultaneously. Explain the difference between the two main approaches to IPL:

Pipelining _____

Multiple Issue _____

2c) What can be done to overcome Amdahl's Law (hint: Gustavson's law)?

_____

2d) What is one-sided communication in the context of distributed memory computing?

_____

2e) If Speedup S = Tserial / T parallel, and P is the number of processors, then what is the formula for Efficiency?

$$E = \frac{T_{serial}}{\text{----------}}$$

2f) MPI is considered (circle one) I) SIMT II) SIMD III) MIMD IV) SPMD

Explain the above acronym chosen: _____

2 g) When can you use OpenMPI, but not OpenMP?

_____

3. Multiple Choice (one or more may be correct) and Short answer (14%)

3a) CUDA: Consider the following code snippets, which are part of CUDA kernels, where n threads each read n values from an n × n array. Which will be faster? (Circle answer)

I)
```
int sum = 0;
for(int i = 0; i < n; i++){
   sum += array[i*n + thread_id];
}
```

II)
```
int sum = 0;
for(int i = 0; i < n; i++){
   sum += array[n*thread_id + i];
}
```

III) They are the same

3b) Circle the following schemes that protect access to critical sections:

I) Semaphores **III) Do-while loops**

II) Mutex Locks **IV) Busy-waiting**

3c) What is the main advantage of read-write locks?

_____

3d) Which of the following schemes are used to reduce branching?

I) Hoisting most frequent case to top or out in separate if-statement

II) Removing branches with labels

III) Removing branches with masks

IV) Memory coalescing (GPUs)

V) Memory striding

VI) All of the above

3e) For each of the following code snippets, determine if the access pattern exhibits spatial locality, temporal locality, both, or neither. Circle accordingly. (a, b and c are in all cases appropriately sized int arrays):

I)
```
for(int i = 0; i < 10000; i++){
  a[i] = b[i] + c[i];
}
```
Spatial / Temporal

II)
```
for(int i = 0; i < 100; i++){
  for(int j = 0; j < 100; j++){
    a[j] = b[j] + c[j];
  }
}
```
Spatial / Temporal

III)
```
for(int i = 0; i < 100; i++){
  for(j = 0; j < 10; j++){
    a[j*1000] += b[j * 2000] + c[j * 3000];
  }
}
```
Spatial/ Temporal

3f) What is thread divergence (hint: GPU)?

_____

3g) Which of the following code snippets will (if executed on a GPU) cause branch divergence? (Circle YES if they do, NO otherwise)

I)
```
if(blockIdx.x > 16){
  foo();
}
else{
  bar();
}
```
YES / NO

II)
```
if(threadIdx.x > 16){
  foo();
}
else{
  bar();
}
```
YES / NO

c)
```
for(int i = 0; i < threadIdx.x; i++){
  foo();
}
```
YES / NO

4. More on MPI (6%)

4 a) Consider the following code, where each rank sends a value to two other ranks, and receives a value from two other ranks. The two ranks that each rank should communicate with, are stored in the variables `n1` and `n2`, and are arbitrary, and specified by the user at runtime.

```
MPI_Send(data_send1, N, MPI_INT, n1, 0, MPI_COMM_WORLD);



MPI_Recv(data_rcv1, N, MPI_INT, n1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);



MPI_Send(data_send2, N, MPI_INT, n2, 0, MPI_COMM_WORLD);



MPI_Recv(data_rcv2, N, MPI_INT, n2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

Can this code potentially cause a deadlock? Circle YES / NO

4 b) If you circled YES to the question above, rewrite the code so that it will never deadlock, using the following MPI functions. Not all the listed functions may be required/are relevant.

Do not use any other MPI functions, but the ones listed below.

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Wait ( MPI_Request *request, MPI_Status *status)

int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request )

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Request *request )

int MPI_Issend( void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request )

int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int
dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
source, int recvtag, MPI_Comm comm, MPI_Status *status )
```

4b) contin: Repeating code to re-write and leaving space for re-write:

```
MPI_Send(data_send1, N, MPI_INT, n1, 0, MPI_COMM_WORLD);

MPI_Recv(data_rcv1, N, MPI_INT, n1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

MPI_Send(data_send2, N, MPI_INT, n2, 0, MPI_COMM_WORLD);

MPI_Recv(data_rcv2, N, MPI_INT, n2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

and leaving space for re-write using only the functions listed on the previous page:

4 c) Consider the following lines, extracted from an MPI program:

```
int* array = malloc(sizeof(int) * M * N);

for(int i = 0; i < M; i++){
     array[i*N + 2] = 10;

}

...

MPI_Datatype new_type;

MPI_Type_vector( ____________________________________________);

...

MPI_Send(array, 1, new_type, dest, tag, MPI_COMM_WORLD);
```

Complete the code above by filling in the arguments to MPI_Type_vector above, so that the MPI_Send will send all the elements set to 10 by the for loop. The definition of MPI_Type_vector is:

```
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype
old_type, MPI_Datatype *newtype_p);
```

5. OpenMP (5%)

Consider the following code, which computes the sum of the elements in an array:

```
int sum = 0;
for(int i = 0; i < N; i++){
     sum += array[i];
}
```

Show how the code can be parallelized using OpenMP. **The resulting code should achieve good parallel speedup.** The following OpenMP directives and functions can be useful:

```
     structured-block
clause:
```

Useful OpenMP directives and functions:

```
#pragma omp parallel [clause[ [, ]clause] ...]
     if(scalar-expression)
     num_threads(integer-expression)
     default(shared | none)
     private(list)
     firstprivate(list)
     shared(list)
```

```
     copyin(list)
     reduction(reduction-identifier: list)
     proc_bind(master | close | spread)


#pragma omp for [clause[ [, ]clause] ...]
     for-loops
clause:
     private(list)
     firstprivate(list)
     lastprivate(list)
     reduction(reduction-identifier: list)
     schedule(kind[, chunk_size])
     collapse(n)
     ordered
     nowait

#pragma omp parallel for [clause[ [, ]clause] ...]
     for-loop
clause: Any accepted by the parallel or for directives,
     except the nowait clause, with identical meanings and
restrictions.

#pragma omp single [clause[ [, ]clause] ...]
     structured-block
clause:
     private(list)
     firstprivate(list)
     copyprivate(list)
     nowait

#pragma omp critical [(name)]
     structured-block


#pragma omp atomic [read | write | update | capture]
expression-stmt

int omp_get_num_threads(void);
int omp_get_thread_num(void);
```

5 CONTINUED: Show below how the code for summing the elements of an array shown earlier can be parallelized using OpenMP. The resulting code should achieve good parallel speedup: (See previous page for useful OpenMP functions)

6. Pthreads (6%)

For each of the following code snippets, determine if it will cause a deadlock if executed by more than 3 threads in parallel. Thread id is an integer storing the id of the thread which is a unique number between 1 and the total number of threads.

a) The following (circle) MAY / MAY NOT deadlock with >3 threads:

```
if(thread_id % 2 == 0){
      pthread_mutex_lock(&mutex_a);
      pthread_mutex_lock(&mutex_b);
      a++;
      b++;
      pthread_mutex_unlock(&mutex_b);
      pthread_mutex_unlock(&mutex_a);
}
else{
      pthread_mutex_lock(&mutex_b);
      pthread_mutex_lock(&mutex_a);
      a++;
      b++;
      pthread_mutex_unlock(&mutex_a);
      pthread_mutex_unlock(&mutex_b);
}
```

6 b) The following (circle) MAY / MAY NOT deadlock with >3 threads:

```
if(thread_id % 2 == 0){
      pthread_mutex_lock(&mutex_a);
      pthread_mutex_lock(&mutex_b);
      a++;
      b++;
      pthread_mutex_unlock(&mutex_a);
      pthread_mutex_unlock(&mutex_b);
}
else{
      pthread_mutex_lock(&mutex_a);
      pthread_mutex_lock(&mutex_b);
      a++;
      b++;
      pthread_mutex_unlock(&mutex_a);
      pthread_mutex_unlock(&mutex_b);
}
```

6c) The following (circle) MAY / MAY NOT deadlock with >3 threads:

```
      if(thread_id % 3 == 0){
            pthread_mutex_lock(&mutex_a);
            pthread_mutex_lock(&mutex_c);
            a++;
            c++;
            pthread_mutex_unlock(&mutex_c);
            pthread_mutex_unlock(&mutex_a);
```

```
}
else if( thread_id % 3 == 1){
      pthread_mutex_lock(&mutex_b);
      pthread_mutex_lock(&mutex_a);
      a++;
      b++;
      pthread_mutex_unlock(&mutex_a);
      pthread_mutex_unlock(&mutex_b);
}
else{
      pthread_mutex_lock(&mutex_c);
      pthread_mutex_lock(&mutex_b);
      c++;
      b++;
      pthread_mutex_unlock(&mutex_b);
      pthread_mutex_unlock(&mutex_c);
}
```

7. OpenCL – 4%

Consider the following OpenCL kernel:

```
__kernel__ work()
int x = get_global_id(0);
int y = get_global_id(1);

if(x % 8 > 4){
func1();
}
else{
func2();
}
}
```

Which is launched like this:

```
size_t global_work_size[2];
global_work_size[0] = 1024; global_work_size[1] = 1024;

size_t local_work_size[2];
local_work_size[0] = m; local_work_size[1] = n;

clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
global_work_size, local_work_size, 0, NULL, NULL);
```

Which of the following values should be used for n and m to minimize divergence when the code is executed on an NVIDIA GPU? (Circle the correct answer.)

I) n = 4, m = 16 II) n = 32, m = 4

III) n = 8, m = 8 IV) n = 4, m = 32

8. CUDA – 10%

In this problem we will look at a CUDA program which computes the average of neighbor elements in an array. In particular, it should perform the following computation on the GPU:

```
out[0] = 0;
out[n-1] = 0;
for(int i = 1; I < n-1; i++){
out[i] = (in[i] + in[i+1] + in[i-1]) / 3.0;
}
```

Thus, if the input is the array [2,3,1,5] the output should be [0,2,3,0], since i.e. (3+1+5)/3 = 3. The elements at the start and end of the output array are set to 0.

In the code given below, the size of the arrays, N, is an arbitrary number larger than 64. The thread block size (i.e. the number of threads in a thread block) is hard coded to be 64. Shared memory is used in an attempt to improve performance, in a similar manner to the way it was done in assignment 6.

The code contains at least 1, and no more than 5 bugs (i.e. 1, 2, 3, 4, or 5 bugs). None of the bugs are syntax errors, i.e. the code will compile without problem, but will crash or produce incorrect results when executed. Ignore problems/bugs related to poor performance, or not freeing memory. The bugs can be in both the device code (i.e. in the kernel) or in the host code, launching the kernel.

Each bug can be fixed by either:

- **Changing a single line.**
- **Removing a single line, and adding a new line somewhere else.**

The code is printed with double line spacing. Correct the bugs by striking out the incorrect lines, and add the corrected version below it, or by striking out a line and adding a new line somewhere else.

Note that in three cases, for the two cudaMalloc calls, and for the final assignment to the out array, one line of code is printed across two lines, but is still regarded as a single line when counting the number of bugs.

```
__global__ void average(float* in, float* out, int N){


     int index = threadIdx.x + blockDim.x*blockIdx.x;

     int lindex = threadIdx.x;
```

```
     __shared__ float shared_array[66];

if(index < N){


     shared_array[index] = in[index];


     if(lindex == 0 && index != 0){


          shared_array[0] = in[index-1];


     }

     if(lindex == 63 && index != N-1){


          shared_array[65] = in[index + 1];

     }

     __syncthreads();

}




if(index == 0 || index == N-1){

     out[index] = 0;

}

else if(index < N){


     out[index] = (shared_array[lindex-1] +
     shared_array[lindex] + shared_array[lindex + 1])/3.0f;
```

```
     }
}// Continues on next page
float* func(int N){

     float* in_host = (float*)malloc(sizeof(float) * N);

     float* out_host = (float*)malloc(sizeof(float)* N);


     float* in_dev;

     float* out_dev;


     cudaMalloc((void**)&in_dev, sizeof(float)*N);

     cudaMalloc((void**)&out_dev, sizeof(float)*N);


     fill_input(in_host);

     cudaMemcpy(in_dev, in_host, sizeof(float)*N,
     cudaMemcpyHostToDevice);

     int nBlocks = N/64;

     if( N % 64 != 0){

          N++;
     }
     average<<<nBlocks, 64>>>(in_dev, out_dev, N);


     cudaMemcpy(out_host, out_dev, sizeof(float)*N,
     cudaMemcpyDeviceToHost);


     return out_host;
}
```

EXTRA PAGE, IF NEEDED. PLEASE INDICATE CLEARLY WHICH PROBLEM you are answering here, if any. May also be used as scratch paper.