PCAP Assignment III

1.

A. **Write a kernel function which takes an array A with *N* (where *N* is multiple of 4) number of integer values as input. It modifies array A by swapping alternate elements (0 with 2, 1 with 3 and so on).**

```
__kernel void swap (__global int *A) {
      size_t i = get_global_id(0) * 4;
      int t = A[i];
      int u = A[i+1];
      A[i] = A[i+2];
      A[i+1] = A[i+3];
      A[i+2] = t;
      A[i+3] = u;
}
```

B. **Write all the statements that are associated with buffer for the above kernel. What is the global work size?**

```
// Create the input and output arrays
cl_mem mem_a = clCreateBuffer(context,  CL_MEM_READ_WRITE,
                              sizeof(int) * count, NULL, NULL);

// Write the data set to the input array in the device memory
clEnqueueWriteBuffer(commands, A, CL_TRUE, 0, sizeof(int) * count,
                              data, 0, NULL, NULL);

// Set the argument to the kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);

// Read the data from the device memory into a output array
clEnqueueReadBuffer(commands, input, CL_TRUE, 0, sizeof(int) * count,
                              results, 0, NULL, NULL);

// Release the memory
clReleaseMemObject(mem_a);
```

The global work size is 256. (Total number of work items)

2.

A. **Explain workgroup and work item concepts by taking the example of a vector of 16 workgroups and 32 work items in each workgroup.**

A kernel dispatch, initiated when the runtime processes the entry in an execution queue created by a call to *enqueueNDRange* on a queue object, consists of a large number of work items intended to execute together to carry out the collective operations specified in the kernel body.

An *NDRange* defines a one, two, or three-dimensional grid of work items.

When mapped to the hardware model of OpenCL, each *work item* runs on a unit of hardware abstractly known as a processing element, called a *work group*, where a given processing element may process multiple work items in turn.

The given example has a total of 16 such processing elements (called *work groups*), and 32 *work items* in each work group, totaling 16 * 32 = 512 *work items*.

B. **How can you find out the execution time taken by kernel function. Specify all the statements required for this.**

We create the command queue with *profiling* enabled;

```
command_queue = clCreateCommandQueue(context, devices[deviceUsed],
                CL_QUEUE_PROFILING_ENABLE, &err);
```

We ensure that all the queue has executed all enqueued tasks

```
clFinish(command_queue);
```

Launch Kernel linked to an event

```
err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
          workGroupSize, NULL, 0, NULL, &event);
```

Ensure kernel execution is finished

```
clWaitForEvents(1 , &event);
```

Get the Profiling data

```
cl_ulong time_start, time_end;
double total_time;

clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
          sizeof(time_start), &time_start, NULL);

clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
          sizeof(time_end), &time_end, NULL);
```

We then get the total time in nano seconds

```
total_time = time_end - time_start;
```

3.

A. **Write a kernel that takes a string *S* and an integer array *Count* consisting of elements equivalent to string length of *S*. It should produce the resultant string *RS* which repeats every character of *S, n* times (where *n* is the integer value in Count which is having the same index as of the character taken in *S)* as shown below.**

Eg :

Input String S	**: string**
Input array Count	**: 1 2 1 3 4 2**
Output String RS	**: sttriiinnnngg**

```
__kernel void multiply (__global char *S, __global int *count,
                                    __global char *RS) {
      size_t id = get_global_id(0);
      int si = 0, i;
      for (i = 0; i < id; ++i) {
            si += count[i];
      }
      for (i = 0; i < count[id]; ++i) {
            RS[si++] = S[id];
      }
}
```

B. **Write all the statements that are associated with buffer for the above kernel. What is the global work size?**

```
char S[DATA_SIZE], RS[DATA_SIZE]; int Count[DATA_SIZE];

cl_mem S_mem = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                              sizeof(char) * DATA_SIZE, NULL, NULL);
cl_mem Count_mem = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                              sizeof(int) * DATA_SIZE, NULL, NULL);
cl_mem RS_mem = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
                              sizeof(char) * DATA_SIZE, NULL, NULL);

clEnqueueWriteBuffer(commands, S_mem, CL_TRUE, 0,
                        sizeof(char) * DATA_SIZE, S, 0, NULL, NULL);
clEnqueueWriteBuffer(commands, Count_mem, CL_TRUE, 0,
                        sizeof(int) * DATA_SIZE, Count, 0, NULL, NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &S_mem);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &Count_mem);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &RS_mem);

clEnqueueReadBuffer(commands, output, CL_TRUE, 0,
                  sizeof(char) * DATA_SIZE, results, 0, NULL, NULL);

clReleaseMemObject(S_mem);
clReleaseMemObject(Count_mem);
clReleaseMemObject(RS_mem);
```

4.

A. **Write an OpenCL kernel code that replaces the principal diagonal elements with zero. Elements above the principal diagonal by their factorial and elements below the principal diagonal by their sum of digits.**

```
__kernel void operate (__global int *A, int count) {
	int i = get_global_id(0);
	int j = get_global_id(1);
	if (i == j) {
		// Primary diagonal
		A[i * count + j] = 0;
	} else if (i < j) {
		// Above diagonal
		int fact = 1;
		for (int k = 2; k <= A[i * count + j]; ++k)
			fact *= k;
		A[i * count + j] = fact;
	} else {
		// Below diagonal
		int z = A[i * count + j];
		int s = 0;
		while (z > 0) {
			s += (z % 10);
			z /= 10;
		}
		A[i * count + j] = s;
	}
}
```

B. **Write an OpenCL kernel code which repeats each element of matrix total element number of times and stores the result in another matrix.**

Sample Input (m * n):

0 1
1 2

Sample Output (m * n * m * n):

0 0 0 0
1 1 1 1
1 1 1 1
2 2 2 2

```
__kernel void operate (__global int *A, int count, __global int *B) {
	int i = get_global_id(0);
	int j = get_global_id(1);
	int x = A[i * count + j];
	int ind = i * count + j;
	int k;
	for (k = 0; k < count * count; ++k) {
		B[ind * count * count + k] = x;
	}
}
```

5.

A. **Write a Parallel OpenCL Kernel code to Replace 1st row of this matrix by same elements, 2nd row elements by square of each element and 3rd row elements by cube of each element and so on.**

```
__kernel void operate (__global int *A, int count) {
      int i = get_global_id(0);
      int j = get_global_id(1);
      int k;
      int x = A[i * count + j];
      int p = x;
      for (k = 1; k <= i; ++k) {
            p *= x;
      }
      A[i * count + j] = p;
}
```

B. **Write a Parallel OpenCL Kernel code to Replace 1st row of this matrix by Elements^(N), 2nd row elements by Elements^(N-1) and N-1 row elements by square of each element Keeping Nth row as it is.**

```
__kernel void operate (__global int *A, int count) {
      int i = get_global_id(0);
      int j = get_global_id(1);
      int k;
      int x = A[i * count + j];
      int p = x;
      for (k = 1; k < count - i; ++k) {
            p *= x;
      }
      A[i * count + j] = p;
}
```