University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2023/2024**
LAB01: Linux Basics	
Student Name:	**Student Reg ID:**

1) Write down a single command that shows the number of available cores on the machine?

2) Write down a command that shows OS kernel version number?

3) Write down a single command that shows the used and free disk space (in gigabytes) available on mounted file systems?

4) Write down a single command that displays the hostname?

5) Write down a single command that lists all files in a directory, including hidden files?

6) Write down a single command that compare two files, i.e., file1.txt and file2.txt line by line while doing case-insensitive comparison?

7) Write down a single command that shows the count of how many images (*.png files) are available in the current directory. You may use pipelining if needed.

8) Write down a single command that shows top 10 processes with highest cpu usage. You may use pipelining if needed.

9) Use vi to create a file text called cutomers.txt and insert the following text:

```
Fred apples 20
Susie oranges 5
Mark watermelons 12
Fred pears 4
Terry oranges 9
Lisa peaches 7
Susie oranges 12
Mark grapes 39
Lisa mangoes 7
Greg pineapples 3
Oliver strawberries 2
Lisa limes 14
```

Now, using pipes and redirection, write down pipe commands needed to find all lines inside fruits.txt that has the customer name “Lisa” and store them into a file called lisa.txt in an <u>alphabetically sorted order.</u>

10) Going back to fruits.txt file in the previous question, write down pipe commands needed to create a new file called customers2.txt that has the same text in customers.txt, except that all letters are capitalized.

University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2022/2023**
LAB02: CUDA Basics	
Student Name:	**Student Reg ID:**

Consider the below serial C program (lab02.c) to complete the following tasks.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define T 100000
#define N 16000

int main(){
 double A[N], B[N];
 double sigma;
 int i;
 double cpu_time;
 clock_t begin, end;

 A[0] = 0;
 A[N-1] = 0;
 for(i=1; i<N-1; i++)
    A[i] = 100.0;

 begin = clock();
 int t = 0;
 while(t++ < T){

   for(i=1; i<N-1; i++)
      B[i] = A[i];

   for(i=1; i<N-1; i++)
       A[i] = (B[i - 1] + B[i + 1])/2.0;

 }
 end = clock();
 cpu_time = (double)(end-begin) / CLOCKS_PER_SEC;

 for(i=0; i<N; i++)
    sigma = sigma + A[i];

 printf("sigma = %.4f\n",sigma);
 printf("cpu time = %.4f\n", cpu_time);
 return 0;
}
```

1) Create an OpenMP version of the given serial code. Name your file lab02_omp.c
   Note the following:
   a) Only focus on parallelizing the for-loops inside the while-loop. Other parts of the code can remain serial.
   b) Use the default static scheduling.
   c) Make sure you keep parallelism overhead at minimal.
   d) To measure execution time, please use omp_get_wtime() function instead of clock() function.

2) Create a CUDA version of the given serial code. Name your file lab02.cu
   Note the following:
   a) Same as above, only focus in the parallelizing the for-loops inside the while-loop, i.e., only these for-loops must be executed on the device. Other parts of the code can be executed on the host.
      Hint: create a separate kernel for each for-loop.
   b) For simplicity, when allocating arrays A and B, use the unified memory model.
   c) To measure execution time, you can keep using the same clock() function already used in the serial version.

3) Now, let us compare performance by filling-in the execution times in the below table. Note that T represents the number of iterations of the while-loop, and N represents the size of arrays A and B. When calculating the execution time for each program, run multiple times and take the average.

	T = 10000
	N = 2048
		T = 100000
		N = 16000

Serial execution time		
OpenMP execution time with 4 threads		
OpenMP execution time with 8 threads		
CUDA execution time with 32 blocks and N/32 threads/block		

University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2022/2023**
Lab03: 2D Arrays in CUDA	
Student Name:	**Student Reg ID:**

Consider the below serial C program (lab03.c) to complete the following tasks.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define T 10000
#define N 800
#define M 400

int main(){
  double A[N][M], B[N][M];
  double sigma;
  int i, j;
  double cpu_time;
  clock_t begin, end;

  for(i=0; i<N; i++)
    for(j=0; j<M; j++){
       if(i==0 || j==0 || i==N-1 || j==M-1)
          A[i][j] = 0.0;
       else
          A[i][j] = 100.0;
    }

  begin = clock();
  int t = 0;
  while(t++ < T){

    for(i=0; i<N; i++)
      for(j=0; j<M; j++)
         B[i][j] = A[i][j];

    for(i=1; i<N-1; i++)
      for(j=1; j<M-1; j++)
         A[i][j] = (B[i-1][j] + B[i+1][j] + B[i][j-1] + B[i][j+1]) / 4.0;

  }
  end = clock();
  cpu_time = (double)(end-begin) / CLOCKS_PER_SEC;

  for(i=0; i<N; i++)
    for(j=0; j<M; j++)
       sigma = sigma + A[i][j];

  printf("sigma = %.4f\n",sigma);
  printf("cpu time = %.4f\n", cpu_time);
  return 0;
}
```

1) Create an OpenMP version of the given serial code. Name your file lab03_omp.c
   Note the following:
   a) Only focus on parallelizing the for-loops inside the while-loop. Other parts of the code can remain serial. Also, because you have a for-loop nest, make sure to parallelize the outer loop.
   b) Use the default static scheduling.
   c) Make sure you keep parallelism overhead at minimal.
   d) To measure execution time, please use omp_get_wtime() function instead of clock() function.

2) Create a CUDA version of the given serial code. Name your file lab03.cu
   Note the following:
   a) Same as above, only focus in the parallelizing the for-loops inside the while-loop, i.e., only these for-loops must be executed on the device. Other parts of the code can be executed on the host.
      Hint: create a separate kernel for each for-loop.
   b) For simplicity, when allocating arrays A and B, use the unified memory model.
   c) To measure execution time, you can keep using the same clock() function already used in the serial version.

3) Now, let us compare performance by filling-in the execution times in the below table. Note that T represents the number of iterations of the while-loop, N represents the number of rows and M represents the number of columns in arrays A and B. When calculating the execution time for each program, run multiple times and take the average.

	T = 10000
	N = 128, M = 64
		T = 10000
		N = 512, M=512

Serial execution time		
OpenMP execution time with 4 threads		
OpenMP execution time with 8 threads		

CUDA execution time with
grid_dims.x = 16
grid_dims.y = 16
block_dims.x = 32
block_dims.y = 32		

University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2023/2024**
Lab04: Parallel Reductions in CUDA	
Student Name:	**Student Reg ID:**

Consider the below serial C program (lab04.c) to complete the below tasks.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 1024
#define M 1024

int main(){
  int **A;
  int i, j;
  int sum = 0;

  A = (int**) malloc(N * sizeof(int*));
  for(i=0; i<N; i++)
      A[i] = (int*) malloc(M * sizeof(int));

  for(i=0; i<N; i++)
      for(j=0; j<M; j++)
          A[i][j] = (int)(0.5 * i + 0.5 * j + 1);

  for(i=0; i<N; i++)
     for(j=0; j<M; j++)
        sum += A[i][j];

  printf("sum = %d\n", sum);
  return 0;
}
```

Create a CUDA version (lab04.cu) of the above serial program. In the CUDA code, you need to have two kernels, as follows:

a) Create a kernel to initialize array A. For this kernel, the number of blocks is N, and the number of threads per block is M.
b) Create a kernel to compute the sum. For this kernel, the number of blocks is N, and the number of threads per block is M/2.
c) **Note**: in both kernels, array A is row-partitioned such that each row is given to a block. This is why the number of blocks is made to be equal to the number of rows.
d) **Note**: in the first kernel, the number of threads per block is M because the j-loop has M independent iterations, i.e., each thread will perform one iteration.
e) **Note**: in the second kernel, the number of threads per block is M/2 because these threads will compute the sum of all M array elements in a row using the algorithm explained in the PowerPoint presentation.
f) In this lab, you are not required to do timing measurements.

University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2022/2023**
LAB06: Java Multithreading	
Student Name:	**Student Reg ID:**

Consider the below Java program, which computes the sum of N floating-point numbers.

```
public class SerialSum {
    static final int N = 100000000 ;

    public static void main(String[] args){

        long st = System.currentTimeMillis();

        double sum = 0;
        for(int i=0; i<N; i++)
            sum += (Math.abs(Math.sin(i)) * 1000);

        long end = System.currentTimeMillis();
        long cpu_time = end - st;

        System.out.printf("sum is %.4f\n", sum);
        System.out.println("cpu time in milliseconds is " + cpu_time);
    }
}
```

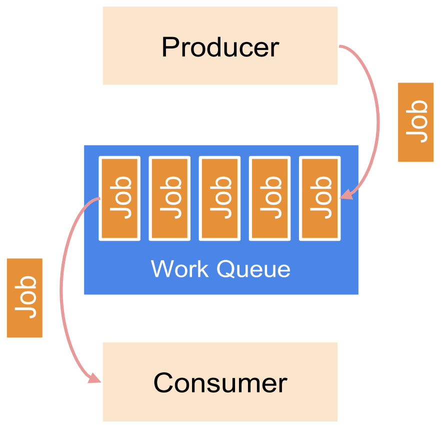
Your task is to create a multi-threaded version of the above serial code. Name your file ParallelSum.java. For generality, Assume T threads in your code, where T is defined as a constant at the beginning of the program, same as N.

Report the execution times (in milliseconds) in the below table.

	N = 100,000,000
Serial	
T = 2	
T = 4	

University of Jordan	**Computer Engineering Department**
Parallel Processing Lab (0907537)	**Fall 2022/2023**
LAB07: Java Multithreading – The Producer/Consumer Problem	
Student Name:	**Student Reg ID:**

In this lab, you are required to implement the producer/consumer pattern in Java (see the picture below). Using a shared work queue, producers add work to the queue, while consumers remove work from the queue. Each producer and consumer are given a unique id. Each job is also given a unique id. To simplify the code, we will assume the job is simply printing a string. In other words, producers add strings to the work queue, while consumers remove these strings from the queue and print them on the screen. Each producer and consumer run as a thread. For CPU efficiency, consumer threads must go to sleep indefinitely when the work queue is empty. Producer threads will notify consumer threads when they add work to the work queue.



In the ***Producer/Consumer*** pattern, one or more ***producers*** creates jobs and adds them to a queue before sending a notification.

One or more ***consumers*** waits for notification. When notified, a ***consumer*** removes the next job from the queue and executes it.

Part 1: The Producer class
Create a class in a file called Producer.java, and do the following:

- Add two attributes: id (int) and workQueue<String> (java.util.LinkedList) and add a constructor that requires both attributes as input arguments
- Make the Producer class extend the Thread class
- In the run method, write code so that the producer adds a string to the work queue every 100 milliseconds and notifies any consumer that is chosen at random to wake up. The first string added to the queue is “job-1”, the second string is “job-2”, the third string is “job-3” and so on. You should use a static counter to keep track of job ids.
- After adding a job to the queue, a producer should print a message on the screen to log this event. For example, if “job-2” has been added by a producer with id=3, then the producer should print “job-2 is added by producer-3”.
- Because the work queue is shared, make sure to use synchronization.
- A producer must run indefinitely so use infinite-loop inside the run method.

Part 2: The Consumer class
Create a class in a file called Consumer.java, and do the following:

- Add two attributes: id (int) and workQueue<String> (java.util.LinkedList) and add a constructor that requires both attributes as input arguments
- Make the Consumer class extend the Thread class
- In the run method, write code so that the consumer goes to sleep when the work queue is empty and only wakes up when notified by a producer to remove a string from the queue. The consumer should then print a message on the screen to log this event. For example, if the removed string is “job-1” and the consumer id is 2, then the consumer should print “job-1 is removed by consumer-2”.
- Because the work queue is shared, make sure to use synchronization.
- A consumer must run indefinitely so use infinite-loop inside the run method.

Part 3: The Test class
Create a class in a file called ProducerConsumerTest.java, and create a main method that instantiate and start five consumer threads and two producer threads. Note all consumer and producer threads must be given the same instance of the work queue.