PCAP Assignment IV

1.

A. Write an OpenCL kernel code which converts input matrix A into output matrix B as follows: consider elements of matrix A into 4 equal parts. First part elements should be incremented by 1, Second part elements should be incremented by 2, Third part elements should be incremented by 3 and last part elements should be incremented by 4. Write the kernel code which does this in parallel for all the elements. Example, N = 4

3	8	2	5
2	3	5	6
2	4	3	1
3	2	1	5

4	9	4	7
3	4	7	8
5	7	7	5
6	5	5	9

```
__kernel void operate (__global int *A, __global int *B, int count) {
            int i = get_global_id(0);
            int j = get_global_id(1);
            // 'd' is the increment value
            int d = 1 + (i >= count/2) * 2 + (j >= count/2);
            B[i * count + j] = A[i * count + j] + d;
}
```

B. Write OpenCL kernel code to calculate the value of Π.

```
__kernel void advance ( __global float *output, const unsigned int
count) {
      int n_rects = get_global_id(0);
      float rect_width = 1.0 / n_rects;
      float rect_left = 0.0;
      float rect_height_squared;
      float rect_height;
      float agg_area = 0.0;
      float pi;
      int i;
      if (n_rects < count) {
            for (i = 0; i < n_rects; i++) {
                  rect_left += rect_width;
                  rect_height_squared = 1 - rect_left * rect_left;
                  if (rect_height_squared < 0.00001) {
                        rect_height_squared = 0;
                  }
                  rect_height = sqrt(rect_height_squared);
                  agg_area += rect_width * rect_height;
            }
            pi = 4 * agg_area;
            output[n_rects] = pi;
      }
}
```

2.

A. **Write a parallel program in CUDA to multiply two Matrices A and B of dimensions MxN and NxP resulting in Matrix C of dimension MxP. Create P number of threads, and each column of the resultant matrix is to be computed by one thread.Use 2D Blocks.**

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void matrix_mult (int *c, int *a, int *b, int ha, int wa) {
      int j = threadIdx.x;
      int wc = blockDim.x;
      int i, k;
      for (i = 0; i < ha; i++) {
            for (k = 0; k < wa; k++) {
                  c[i*wc + j] += a[i*wa + k] * b[k*wc + j];
            }
      }
}

int main () {
      int ha = 2; int wa = 3; int hb = 3; int wb = 2; // Shhhh, it's okay
      int hc = ha;
      int wc = wb;

      int a[6] = { 1, 2, 5, 3, 4, 6 };
      int b[6] = { 7, 10, 8, 11, 9, 12 };
      int c[4] = { 0 };

      int size_a = ha * wa, size_b = hb * wb, size_c = hc * wc;

      // Print A, B

      int *dev_a, *dev_b, *dev_c = NULL;


      cudaMalloc((void**)&dev_a, size_a * sizeof(int));
      cudaMalloc((void**)&dev_b, size_b * sizeof(int));
      cudaMalloc((void**)&dev_c, size_c * sizeof(int));

      cudaMemcpy(dev_a, a, size_a * sizeof(int), cudaMemcpyHostToDevice);
      cudaMemcpy(dev_b, b, size_b * sizeof(int), cudaMemcpyHostToDevice);

      dim3 threadsPerBlock(wc,1);//just for the sake of 2D-Blocks;
      matrixMult <<<1, threadsPerBlock >>>(dev_c, dev_a, dev_b, ha, wa);

      cudaMemcpy(c, dev_c, size_c * sizeof(int), cudaMemcpyDeviceToHost);

      // Print C

      return 0;
}
```

Thanks Jawahar for CUDAing

B. **Compare Superscalar, VLIW and Hardware Multithreading.**

Superscalar machines are able to dynamically issue multiple instructions each clock cycle from a conventional linear instruction stream.

VLIW processors use a long instruction word that contains a usually fixed number of instructions that are fetched, decoded, issued, and executed synchronously.

A "hardware thread" is a physical cpu or core. It can run many software threads.

3.

A. **Write an OpenCL Kernel code to perform odd-even transposition sorting in parallel. What is grid?**

```
__kernel void even (__global int *a) {
      int idx = get_global_id(0);
      int size = get_global_size(0);
      int temp;
      if ((idx % 2 == 0) && ((idx + 1) < size)) {
            if (a[idx] > a[idx+1]) {
                  temp = a[idx];
                  a[idx] = a[idx+1];
                  a[idx+1] = temp;
            }
      }
}

__kernel void odd (__global int *a) {
      int idx = get_global_id(0);
      int size = get_global_size(0);
      int temp;
      if ((idx % 2 != 0) && ((idx + 1) < size)) {
            if(a[idx] > a[idx+1]) {
                  temp = a[idx];
                  a[idx] = a[idx+1];
                  a[idx+1] = temp;
            }
      }
}
```

In OpenCL execution model, the kernel instances execute concurrently over a virtual ***grid*** defined by the host code.

B. **How many grids will be created for odd-even transposition sorting and merge sorting.**

However many you create ¯_(ツ)_/¯.
You can do it in one grid or N grids.

4.

A. **Write an OpenCL Kernel code to perform merge sort in parallel.**

// Copying this one straight off the interwebs. No time for this shit.

```
__kernel int merge(__global const int *arr1, int arr1size, __global const
int *arr2, int arr2size, __global int *out) {
    int x = 0;
    int y = 0;
    int n = 0;

    while(x < arr1size || y < arr2size) {
      if(x < arr1size && y < arr2size) {
        if(arr1[x] < arr2[y]) {
          out[n] = arr1[x];
          n++,x++;
        } else {
          out[n] = arr2[y];
          n++,y++;
        }
      } else if(x < arr1size) {
        out[n] = arr1[x];
        n++,x++;
      } else if(y < arr2size) {
        out[n] = arr2[y];
        n++,y++;
      }
    }
    return (x + y) - (arr1size + arr2size);
}

__kernel void mergesort(__global int *a, __global int *b, __global int *len)
{
    int size = 1;
    int indx = get_global_id(0) * 2;
    int i = 0;

    while(size <= *len) {
      barrier(CLK_LOCAL_MEM_FENCE);
      if((indx + size) < *len) {
         i = (indx + size) + (((indx + (size * 2)) > *len) ? *len - (indx +
size) : size);
            merge(&a[indx], size, &a[indx + size], ((indx + (size * 2)) >
*len) ? *len - (indx + size) : size, &b[indx]);
      } else {
        return;
      }
      size *= 2;
      cpy(&a[indx], &b[indx], size);
      if(indx%(2*size) != 0) {
        return;
      }
    }
```

```
    *len = i;
}
```

B. What is data parallelism? Explain With the help of suitable example.

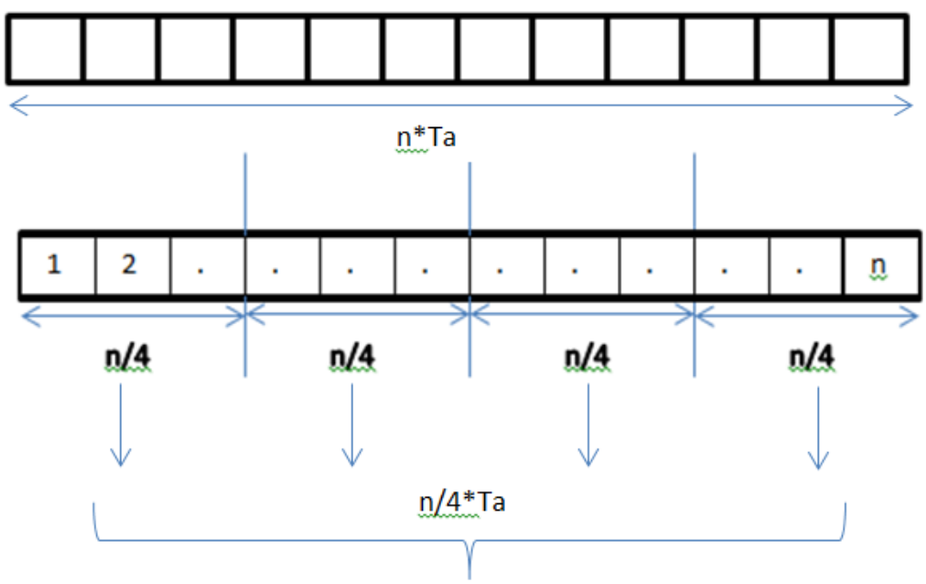
Data parallelism is a form of parallelization across multiple processors in parallel computing environments. It focuses on distributing the data across different nodes, which operate on the data in parallel. It can be applied on regular data structures like arrays and matrices by working on each element in parallel.

Let us assume we want to sum all the elements of the given array and the time for a single addition operation is Ta time units.

In the case of sequential execution, the time taken by the process will be n*Ta time units as it sums up all the elements of an array.

On the other hand, if we execute this job as a data parallel job on 4 processors the time taken would reduce to (n/4)*Ta + Merging overhead time units.

Parallel execution results in a speedup of 4 over sequential execution.



5.

A. **Write a parallel program in CUDA to add two Matrices A and B of dimensions MxN and NxP resulting in Matrix C of dimension MxP. Create P number of threads, and each column of the resultant matrix is to be computed by one thread. Handle errors.**

```
// Shhhh, just okay.
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__global__ void matrixAdd(int *c, int *a, int *b, int height) {
	int j = threadIdx.x;
	int k = blockDim.x;
	int i;
	for (i = 0; i < height; i++)
		c[i*k + j] = a[i*k + j] + b[i*k + j];
}

void checkForError(cudaError_t e) {
	if (e != cudaSuccess) {
		printf("Error : %s\n", cudaGetErrorString(e));
		exit(1);
	}
}

int main () {
	int height = 2;
	int width = 2;

	int a[4] = { 1, 2, 3, 4 };
	int b[4] = { 1, 2, 3, 4 };
	int c[4] = { 0 };

	int *dev_a = NULL, dev_b = NULL, dev_c = NULL;

	int size = height * width;

	cudaError_t e;

	e = cudaMalloc((void**)&dev_c, size * sizeof(int));
	checkForError(e);

	e = cudaMalloc((void**)&dev_a, size * sizeof(int));
	checkForError(e);
	e = cudaMalloc((void**)&dev_b, size * sizeof(int));
	checkForError(e);

	e = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
	checkForError(e);
	e = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
	checkForError(e);
```

```
    matrixAdd <<<1, width >>>(dev_c, dev_a, dev_b, height);
    e = cudaGetLastError();
    checkForError(e);

    e = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
    checkForError(e);

    int i;
    for (i = 0; i < size; i++) {
        if (i%width == 0)
            printf("\n");
        printf("%d ", c[i]);
    }
    return 0;
}
```

B. Compare different parallel programming models.

Shared Memory Model

In a shared-memory model, parallel processes share a global address space that they read and write to asynchronously. Asynchronous concurrent access can lead to race conditions and mechanisms such as locks, semaphores and monitors can be used to avoid these.

Message Passing Model

In a message-passing model, parallel processes exchange data through passing messages to one another. These communications can be asynchronous, where a message can be sent before the receiver is ready, or synchronous, where the receiver must be ready.

Implicit Interaction Model

In an implicit model, no process interaction is visible to the programmer and instead the compiler and/or runtime is responsible for performing it. Two examples of implicit parallelism are with domain-specific languages where the concurrency within high-level operations is prescribed, and with functional programming languages because the absence of side-effects allows non-dependent functions to be executed in parallel.