GPU Programming Assignment - 3
AE13B064

Question - 1

Since there is no `__syncthreads()` operation before the `printf()` 3 cases are possible. 2 warps are possible one with 32 active threads and the other with 2 active threads

Case 1: Both warps finishes the job of increment simultaneously, in this case all there will be 34 - 34's in the output screen.

Case 2: Warp 1 finishes the job (first 32 threads) before warp 2, in this case there will be 32 - 32's and 2 - 34's in the output screen.

Case 3: Warp 2 finishes the job (2 active threads) before warp 1, in this case there will be 2 - 2's and 32 - 34's in the output screen.

Question - 2

Since coalescing is access of memory which is used by the warp threads at single time step, the following code snippet achieves various degrees of coalescing by accessing useful memory for warp threads in different time steps (depending on the degree).

So, the idea for coalesced access of an array (of size 32 for example) is to find the number of time steps required to access the array elements, such that when parameter passed is 1 elements are accessed in 32 steps. Similarly if parameter 2 is passed elements are accessed in 31 time steps. Finally if parameter 32 is passed elements are accessed in a single time step.

```


__global__ void coalescing(int * arr, int param) {
  int id = threadIdx.x;
  int timeSteps = 32 - param + 1; // Offset which is required to calculate how many  access at a time

  for(int i = 0; i < timeSteps; i++){
    if(id >= i and id < param + i){ // Checks if the thread id is with in the range for accessing memory from array at that particular time Step
       arr[id]++; // Process something with the fetched memory
    }
```

```
  }
}
```

Listing 1: Degree of coalescing Kernel

Testing the Code

The code is tested with following `main.cu` where the kernel call is run for 10000 times to find the average time it took to finish the kernel for various degrees of coalescing.

```
#include <cuda.h>
#include <stdio.h>
#include "timer.h"
#include "kernel.h"

int main() {
    int N = 32; // Array of size 32
    int *counter;
    int *hcounter = (int*)malloc(N*sizeof(int));
    int i = 0;
    for(i = 0; i< N; i++){
      hcounter[i] = i;
    }

    cudaMalloc(&counter, N*sizeof(int));
    cudaMemcpy(counter, &hcounter, sizeof(int),
    cudaMemcpyHostToDevice);
    GPUTimer timer;
    float t = 0;
    for(i = 0; i< 10000; i++){
      timer.Start();
      coalescing<<<1, N>>>(counter, 16);
      cudaDeviceSynchronize();
      timer.Stop();
      t += 1000*timer.Elapsed();
    }

    printf("GPU elapsed %f ms \n", t/10000);
    return 0;
}
```

Listing 2: Degree of coalescing Kernel

Experiment is performed for various degrees of coalescing for increment operation of array values. Following are the results:

Degree of Coalescing	Average time for 10000 calls (ms)
1	20.67
16	17.51
32	14.11

Question - 3

Following is the code snippet for finding the saddle point in the kernel. Logic is to assign single row to each thread. Then, find the minimum in that corresponding thread; store the col index of minimum; find the maximum in that particular column where minimum is found; store the row index where maximum is found in that particular column; if the row index matches with thread id then we found our saddle point

```

__global__ void saddleFinder(const int *arr, int N){
  int id = threadIdx.x;
  int max = INT_MIN, min = INT_MAX; // Using limits.h
  int i1 = 0, i2 = 0 ;

  __syncthreads(); // It is not required

  // Finding minimum in a row
  for(int i = 0; i < N; i++){
    if(min > arr[id*N+i] ){
      min = arr[id*N + i]; // Storing the minimum for checking in next iteration
      i1 = i;              // Storing the coloumn where minimum is found;
      //printf("threadIdx: %d, min: %d, i1: %d\n", id, min, i1);
    }
  }

  __syncthreads();// Not required


  // Finding Maximum in the coloumn corresponding to found minimum (in i1)
  for(int i = 0; i < N; i++){
    if(max < arr[i*N + i1]){
      max =  arr[i*N + i1]; // Storing the Maximum
      i2 = i;               // Storing the row where Maximum is found
      //printf("threadIdx: %d, max: %d, i2: %d\n", id, max, i2);
    }
  }

  // Check if the row matches with corresponding thread id (which determines the saddle condition)
  if (i2 == id){
    printf("Saddle Found: %d, at (%d,%d)\n" , arr[id*N + i1], id, i1 );
  }
}
```

Listing 3: Saddle Finder Kernel

Testing the code

The code is tested with following matrix `main.cu`

```
int main(){
  // specify the dimensions of the input matrix
  const int M = 3; // square matrix dimensionss
  unsigned numbytes = M * M * sizeof(int);

  int arr[M][M] = {{1,2,4},{-1,-2,2},{4,5,6}}; // 3x3 matrix; Saddle at (2,0) == 4

  int *out;
  cudaMalloc(&out, numbytes);
  cudaMemcpy(out, arr, numbytes, cudaMemcpyHostToDevice);

  saddleFinder<<<1,M>>>(out, M);
  cudaDeviceSynchronize();
  return 0;
}
```

Listing 4: Saddle Finder Main