Links: GITHUB: https://github.com/VinitSaah/WES237B/tree/main/Assignment4

Lab
Part 1

1. Example 1 (ex1.cu) initializes an integer array a of size 4. It decleares a pointer dev_a. dev_a is allocated using cudaMalloc with the size of array times size of int. cudaMalloc allocates linear memory on device and returns a pointer to host. cudaMemcpy is then used to copy data from host's integer a to device on area marked by pointer dev_a. Number of blocks, b, is initialized to 2, Number of threads,t, per block is initialized to 2. Kernel function myKernel is launched using <<<b,t>>>(dev_a). It basically launches 4 copies of function myKernel. All thread executes the function. myKernel function prints the corresponding block Id, thread Id. The function uses standard approach of determining global thread id associated with the thread. Finally, host function waits for all threads to finish executing using cudaDeviceSynchronize. Allocated memory on the device is freed by the host using cudaFree.

2. Example 2 (ex2.cu) is similar to ex1.cu with only difference being the block and thread assignment. Ex2.cu initializes two blocks with two threads each. In total the threads are 4.
3.

```
__global__ void myKernel(int *m, int *v, int *r){
  // write your code here
  uint thread_global_idx = blockIdx.x * blockDim.x + threadIdx.x;
  //printf("block[%d], thread[%d]: m[%d]=%d\n", blockIdx.x, threadIdx.x, thread_global_idx, m[thread_global_idx]);
  r[thread_global_idx] = 0;
  for(int i =0; i < 3;i++)
  {
    r[thread_global_idx] += m[(thread_global_idx*3)+i]*v[i];
  }
}
```

4. lw.cu implementation: does a matrix multiplication over memory allocated on Device using cudaMalloc. It copies the matrices on device using cudaMemcpy from host. When kernel execution is over by all threads, the host (main program) copies the result matrix pointed by dev_r back to host array r using cudaMemcpy.
   However, in the other lw_managed.cu, both host and device share common memory called as unified memory managed by unified memory system. This eliminates the need of doing cudaMemcpy from host to device and device to host and hence reducing the lines of code.

Part 2

Input image size (1024,1024)
Blur Size = 10;

For Inverting and Blurring: First RGB is converted to Gray and then algo is applied. So, the total average time includes time taken to convert RGB to Gray. Note the time shown below is average of 10 iterations of time to output 10 frames from the output as shown in the example below.

Using OpenCV Functions
Execution Time (s) = 0.00124879
Execution Time (s) = 0.00151691
Execution Time (s) = 0.00144109
… … … …
Average = 0.00148642 Sec

OpenCV output

Serial Number	Algorithm	Execution Time (s)
1	RGB to Gray	0.00093219
2	Invert	0.00148642
3	Blur	0.00470603

CPU Output

Serial Number	Algorithm	Execution Time (s)
1	RGB to Gray	0.00826455
2	Invert	0.00940077
3	Blur	0.2707736

GPU Output Non-Unified Memory

Serial Number	Algorithm	Execution Time (s)
1	RGB to Gray	0.00636299
2	Invert	0.00942369
3	Blur	0.02338549

GPU Output Unified Memory

Serial Number	Algorithm	Execution Time (s)
1	RGB to Gray	0.00373646
2	Invert	0.00585832
3	Blur	0.02066322

OpenCV is outperforming all the cases. Shortest time taken is to convert RGB to Gray. In GPU, the Unified memory is taking less time.

To test when GPU comes closer to OpenCV output, increased size of image input to 4096 * 4096.

The RGB to GRAY conversion showed following timings.

OpenCV: 0.01059199 s
GPU Unified: 0.0200897 s

Assignment

Part 1: Sobel Filter

I used UNIFIED approach for this experiment. However, the code has an option to disable the Unified approach.

Input Image Size : 512,768, 1024, 2048, 4096. Used unrolled approach for Sobel filter.

OpenCV Sobel output

Image Size	512	768	1024	2048	4096
Execution Time(ms)	8.068042	17.0241091	29.2159455	108.692	444.357636

CPU Output

Image Size	512	768	1024	2048	4096
Execution Time(ms)	3.64280182	7.86719273	13.6205273	52.6113545	203.577091

GPU Output

Image Size	512	768	1024	2048	4096
Execution Time(ms)	1.45831273	2.89269091	4.95062364	19.4187909	23.0839364

GPU output is way fast than OpenCV and CPU Output.

Note, the given Sobel filter executable runs with non-square sizes, try testing out with 511x513 or 789x441

Part 2: Blocked matrix multiplication

Expected RMSE: The RMSE should be small, below **0.001**

2) Block Size taken = 16

Matrix Multiplication

Case 1, M < N

M = 32, 64, 128, 256, 512,1024
N = 1024, 2048

S no	Block Size	M	N	CPU (ms)	GPU (ms)	Speed up	RMSE
1	16	32	1024	39.4	10.43	3.79x	0.00000
2	16	64	1024	83.37	18.31	4.55x	0.00000
3	16	128	1024	194.82	36.89	5.28x	0.00001
4	16	256	1024	401.50	71.74	5.60x	0.00001
5	16	512	1024	773.87	113.88	6.80x	0.00004
6	16	1024	2048	6080.16	229.68	26.47x	0.00011

Case 2, M = N

M = 1024, 2048, 4096,4112
N = 1024, 2048, 4096,4112

S no	Block Size	M	N	CPU (ms)	GPU (ms)	Speed up	RMSE
1	16	1024	1024	1556.77	105.25	14.79x	0.00011
2	16	2048	2048	12141.60	384.29	31.59x	0.00031
3	16	4096	4096	102688.93	2640.07	38.90x	0.00086
4	16	4112	4112	100636.60	2669.8	37.69x	0.00087

Case 3, M > N

M = 1024,2048
N = 32, 64, 128, 256, 512,1024

S no	Block Size	M	N	CPU (ms)	GPU (ms)	Speed up	RMSE
1	16	1024	32	2.27	0.69	3.27x	0.00011
2	16	1024	64	10.44	1.39	7.53x	0.00011
3	16	1024	128	27.94	4.64	6.02x	0.00011
4	16	1024	256	100.32	17.87	5.61x	0.00011
5	16	1024	512	392.73	71.58	5.49x	0.00011
6	16	2048	1024	3213.52	395.92	8.12x	0.00031

From all the cases it could be seen that speed increases when sizes of M and N increases for a fixed block size. However, for case 1024*256 and 1024* 512, it could be seen that did not increase from the previous size.

3) Finding the proper block size.

Let the input size be 4096 * 4096. Experimental block sizes 4, 8, 16, 32. Could not go bigger as it gave setup error.

S no	Block Size	M	N	CPU (ms)	GPU (ms)	Speed up	RMSE
1	4	4096	4096	102105.18	23690.03	4.31x	0.00086
2	8	4096	4096	96680.16	3954.71	24.45x	0.00086
3	16	4096	4096	102452.07	2867.22	35.73x	0.00086
4	32	4096	4096	101330.82	2458.32	41.22x	0.00086

From experiments, Block Size found is 32. This means, the shared square matrix size would be **32x32**.

Number of Blocks would be 32. And Grid dimension would be 128.