GPU LAB-3

Name: Riddhi R. Thakker
ID: 201601124
Name: Purva Mhasakar
ID: 201601082

QUESTIONS

1. **How many floating operations are being performed in your color conversion kernel?**
   ANSWER:
   The threading is done on the basis of the output gray image, for each output pixel one thread is allocated. For this problem, we require 2D block allocation and, hence the mapping is done using following logic:
   int x = threadIdx.x + blockIdx.x * blockDim.x
   int y = threadIdx.y + blockIdx.y * blockDim.y
   int idx = x*blockDim.y + y
   if(idx¡imageWidth*imageHeight)
   r = colorImage[3*idx]
   g = colorImage[3*idx + 1]
   b = colorImage[3*idx + 2]
   grayImage[idx] = (unsigned char)((21*r + 71*g + 7*b)/100)
   This alone will require 6 floating operations for each thread. Further, for computing the gray code of a pixel, we will require to access 3 values: r, g, b of color image and in total this will result into additional 12 floating operations.
   Total floating operations in the kernel: 18 Flops * (N*M) Threads

2. **How many global memory reads are being performed by your kernel?**
   ANSWER:
   For every gray value calculation, we require 3 values: r, g, b of color image. Hence, total global memory reads: 3 * (N*M) Threads

3. **How many global memory writes are being performed by your kernel?**
   ANSWER:
   The size of the gray image is N*M and each pixel has only one value. Hence, total global memory writes: N*M

CONTEXT

1. **Description of the problem**
   Image Conversion from color to gray:
   The problem is to convert RGB colored image into gray image.

2. **Complexity of Serial Code:**
   $\mathcal{O}(N * M)$
   where NxM is the size of the gray image.

3. **Possible Speedup (theoretical):**
   Theoretical Speedup can be defined using Amdahl's Law:

$$S_{Latency} = \frac{1}{(1-p) + \frac{p}{s}}$$

   where:
   $S_{Latency}$ = theoretical speedup
   s = speedup of the part of the task that is parallelized
   p = proportion of execution time that the part benefiting from parallelism

4. **Optimization Strategy:**
   Following factors are to be considered while paralleling a code:

   (a) Dependency:
   The operation of color to gray is done on each pixel, independently. Hence, there is NO dependency among the computations and can be parallelized easily on each pixel of the image.

   (b) Locality of Reference (or Re-usability):
   Here, a particular memory location is accessed only once and hence, there is NO re-usability i.e. temporal locality is not present. However, there is considerable spatial locality present in this program but this property of the data can be utilized only if we are using shared memory.

   (c) Optimization (Scope of parallelization):
   The memory accesses which are done serially can be parallelized. But, the number of computations required remains the same.

   (d) CGMA (Compute to Global Memory Access):
   As we are not parallelizing the computation part, the number of computations remains same as they were in serial code. Only the net time to memory access decreases, and not the number of memory accesses required. Hence, throughput will increase but CGMA will remain unchanged. CGMA can be increased by using shared memory.

   (e) Synchronization:
   The programs are free from all kind of dependencies which lead to asynchronization (such as: flow, anti, input, output, loop carried). Hence, synchronization between the threads is not required in this case.

   (f) Control Divergence:

The thread organization is 2 dimensional, hence, there are high chances that control divergence will occur among some warps. Therefore, while deciding the block dimensions, this parameter has to be taken into consideration.

(g) Thread Organization:
Here it is better if we go for 2 dimensional thread organizaion as the mapping among the indices and the threads become easy.

5. **Problems in parallelization and possible solutions:**
The main important problem while parallelizing is the identify a pattern and then, verifying the dependencies among them. Afterwards, the thread organization is another problem, which has to be solved considering the hardware properties of the device. It is necessary to ensure that there is as less as possible control divergence presence. The resource allocation should satisfy the hardware limitations otherwise the code will not execute and there will be errors in the output.

OBSERVATIONS

1. Comparison of Serial & Parallel Code

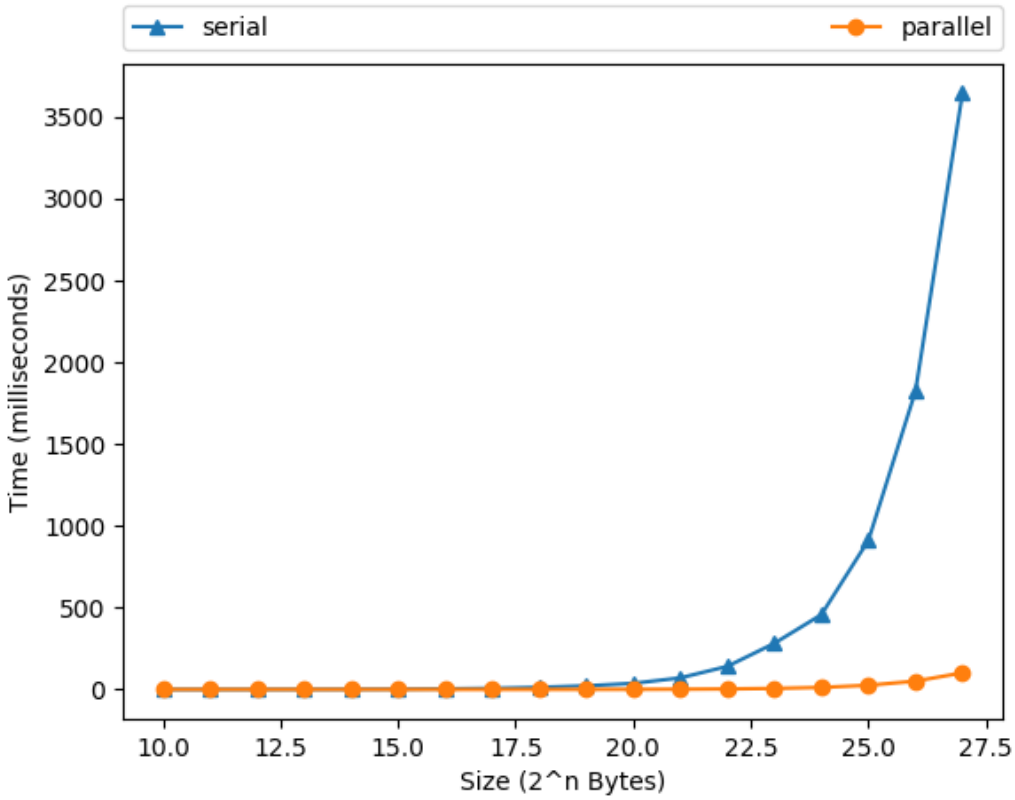

Figure 1: Time VS Problem Size

The time taken by serial code increases exponentially after 2^{20} where as time taken by parallel code remains almost same throughout the experiment.
The memory accesses which were serial are now parallel, hence the overhead of time for global memory access is parallelized thus reducing the net time for global memory access as compared to one in serial.

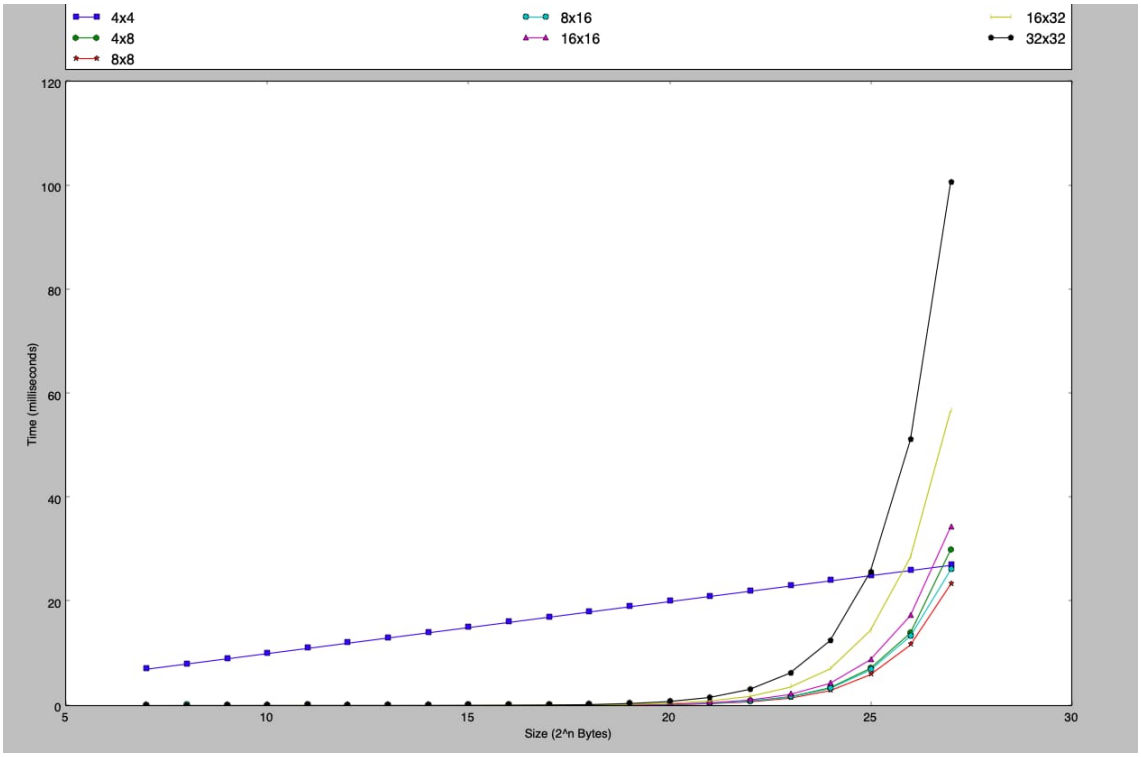

Figure 2: Time VS Problem Size for different block sizes

The time taken by 4x4 block dimension is more than any other choices of block dimension, because 4x4 includes 16 threads in one block and there are 32 threads per warp, so there will be control divergence in all the blocks that are required for a particular size of image.
From this above figure, it is clear that after 2_{18} problem size, the time taken by 32x32 block size increases considerably. The architecture on which the cuda code is running is Kepler GTX 680, which allows 1024 threads/block, 8 blocks/SM and 2048 threads/SM at a given point of time. There are 1536 CUDA cores, and with 32 CUDA cores in one SM, there are total $1536/32 = 48$ SMs available at a point of time. And with 32x32 block size, we can only launch 2 blocks in one SM due to the 2048 threads/SM constraint. Thus, in total at any given point of time, we can launch $2 * 48 = 96$ blocks at a time. Upto problem size 2_{16}, we required blocks less than 96 but after that at problem size 2_{17}, we required total 128 blocks to schedule but due to the hardware constraint, scheduler will launch 96 blocks at a time and will schedule other $(128 - 96 =)$ 32 blocks afterwards together (since $32 < 96$). Thus, for 1 single flop to complete in all the threads we need potentially 2 phases of scheduler. The number of phases required for this will keep on increasing as the problem size

increases as the number of required blocks also increases. For example, we require 256 blocks for problem size 2_{18}, which can be broken down into $2 * (96) + 64$, further for problem size 2_{22}, we require 4096 blocks which can be broken down into $43 * (96) + 64$.

2. Speedup Curve

   $Speedup = SerialTime/CUDATime$

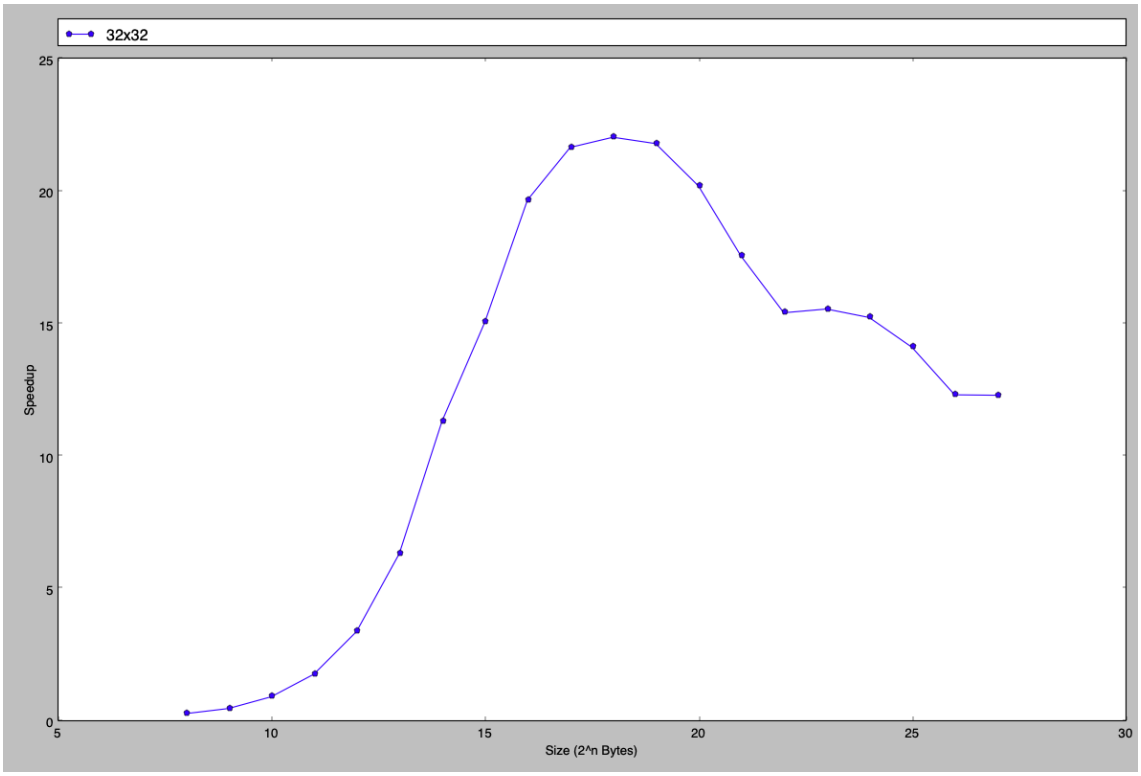

Figure 3: Speedup VS Problem Size for 32x32 block size

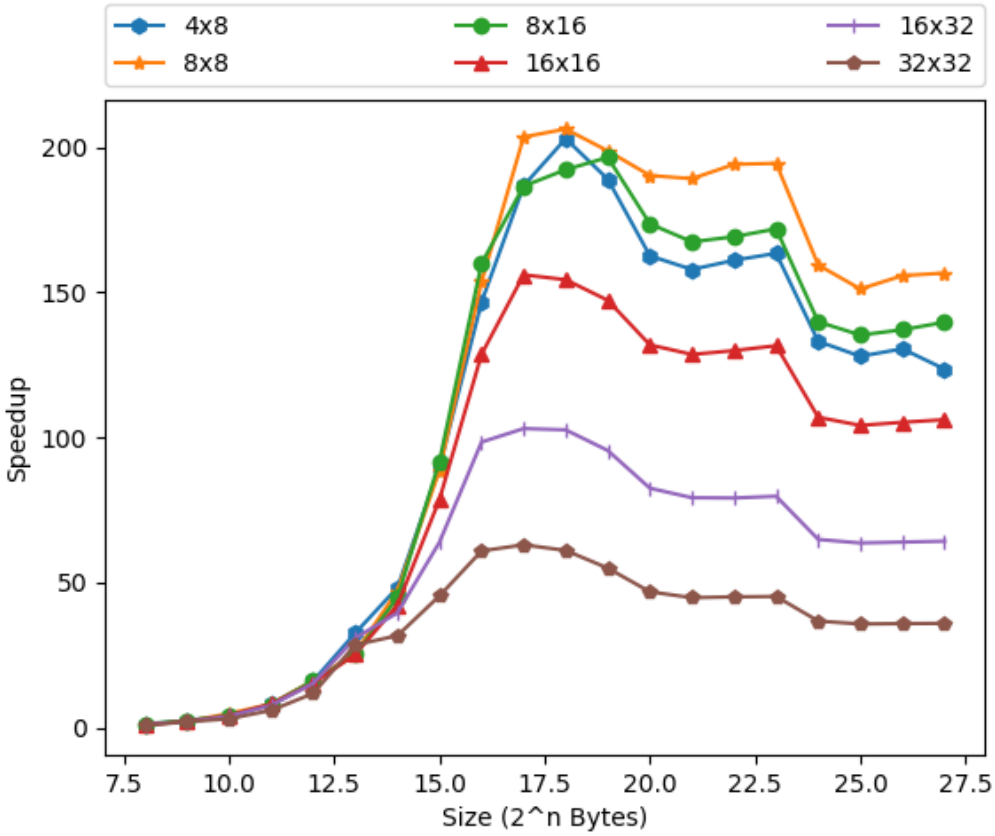

Figure 4: Speedup VS Problem Size for different block sizes

3. Effect of Block Size on Speedup
   From the figure 3, it is observed that speedup increases as the problem size increases upto 2_{18} and later on decreases. Thus, showing a peak speedup at problem size 2_{18}. The reason for such a graph is as below: The architecture on which the CUDA code is running is Kepler GTX 680. There are in total 1536 CUDA cores and thus (1536/32 =) 48 SMs available at a given time. Other constraints are 1024 threads/block, 8 blocks/SM and 2048 threads/SM. Now, with these constraints we can launch only 2 block of size 32x32 at a time on one SM. Thus totaling to $2 * 48 = 96$ block at a time and the rest will be scheduled by the warp scheduler. Upto problem size 2_{16}, we require lesser number of blocks than 96 but afterwards we required more than available blocks. Henceforth, time taken by 1 flop operation to be complete in all the threads will be more as compared to lesser problem size. Though, the threads with run in parallel to each other and according to the resources available they will execute in parallel, but the synchronization between the threads will

not be maintained in terms of computation which justifies the fact that we require more time to complete 1 flop operation in all the threads. The speedup still increases upto problem size 2_{18} because the ratio of threads executing the same instruction to that of different is still less as compared to later problem sizes. Hence, the time taken by the parallel code is still considerably less than that of serial code. As the problem size increases, the time taken by the parallel code increase due to the reason specified (shown in the figure 2 and explained there), the speedup decreases.

This is also the reason why the speedup curve (figure 4) for 8x8 block size is way higher than 32x32 whereas in later part we are utilizing the complete block dimension available to us. Because, we can launch more number of blocks per SM now, as the threads/SM constraint is now weaken in case of 8x8 block size. This is the main reason why the speedup curve of 8x8, 8x16, 16x16, 16x32 block size are higher as compared to 32x32.

The other reason could possibly be that the GPUs are designed to fast the computations done and are not that efficient in memory accesses as compared to CPU. But, here in this code, the scope of parallelism is limited to the memory accesses that are done serially in CPU and the computation part is not touched upon. Hence, the speedup that we are obtaining are mainly due to the parallelization of memory accesses and not the computation part.

NOTE:

1. **The test images for this code are in the following link:**
   `https://drive.google.com/file/d/1AY9P32cybphLFxI73cQHKShwaLp7MU0S/view?usp=sharing`
   These images are to be placed in the same folder as of the code files.

2. time_analysis_32x32_block.png : this is the time vs problem size comparison among serial and parallel codes of image conversion (rgb to grayscale).

3. speedup_analysis_blocks.png : this is the speedup vs problem size comparison for different block sizes considered for results.

4. time_CD_analysis_blocks.png : this graph has one value where control divergence is observed, this Control divergence is implicitly implemeted to compare its result with other thread organization choices.

5. time_analysis_blocks.png : this is the time vs problem size comparison between different block sizes chosen, control divergence is not included among the choices of block size in this graph.

6. serial_image.c : this is the C code for converting RGB image to grayscale image. The size of the images goes from 2_7 to 2_{27} (both included).

7. parallel_image.cu : This is CUDA code for the same purpose with fixed block size of (32x32).

8. block_parallel.cu : This CUDA code computes time required to carry out parallel grayscale conversion for different image sizes with varying block sizes. Block sizes varies from 2_5 to 2_{10}. We started from 2_5 because a warp consists of 32 threads, so the minimum possible thread organization would be 32, and the maximum would be equal to that of the max size of the threads per block (i.e. 1024).

9. block_CD.cu :This CUDA code simply considers control divergence by starting the block sizing from 2_4.