**Question 1**

How many different dimensions of parallelism exist in the histogram application? What decomposition pattern does each dimension fit best? Please describe in detail where you found concurrency and your reasoning for choosing the patterns.

**Answer 1**

There are two different levels of parallelism in the application

1. Pixels: Histogram calculation for each of the pixels is independent of the other pixel. Essentially this can represent one level of parallelism.
2. Colors: The three colors Red, Green and Blue are also independent of each other and can be worked on in parallel. Essentially for each of the pixel we can read the value of all the three colors also in parallel.

The part which deals with updating of histogram values in the bins can be concurrent at times. Consider the situation where a particular color in two pixels is having the value. The increment operation in the will have to be sequential in such a scenario. As such though the reading part is parallel this is sequential and as such the whole scenario becomes concurrent.

**Question 2**

How many different general ways can you think of for designing your parallel algorithm? An example of a general way would be to use a mutex to ensure only one update is performed to each bin at any given time.

Please describe each way in words as well as providing pseudocode.

I could come up with 4 general ways, one of which uses mutexes/atomic-operations. Two of the other methods scale fairly well with *N*, *P*, and *B* (i.e., execution time improves significantly compared to serial execution regardless of *N* or *B*). The fourth method is not all that scalable, but could be appropriate when *B* is large. There are probably other methods as well.

**Answer 2**

**Method 1**

A simpler approach would be to update the values in the bins one after the other with the bins in global memory.

\_\_global\_\_ void histogram\_globalMem(int \* red\_bin, int \* green\_bin, int \* blue\_bin){

//pixel coordinates

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

int tIndex = blockDim.x \* threadIdx.y + threadIdx.x;

unsigned int r = (unsigned int)(img[x][y].r/ BIN\_WIDTH);

unsigned int b = (unsigned int)(img[x][y].b/BIN\_WIDTH);

unsigned int g = (unsigned int)(img[x][y].g/BIN\_WIDTH);

atomicAdd(& red\_bin[r], 1);

atomicAdd(& blue\_bin[b], 1);

atomicAdd(& green\_bin[g], 1);

}

The problem with this method is atomic operations are being performed on global memory. Atomic operations on global memory are very slow.

**Method 2**

A slight variation to the above method can be that we have 3 threads per pixel and each of the thread works on one of the colors. This method utilizes both level of parallelism.

\_\_global\_\_ void histogram\_globalMem(int \* red\_bin, int \* green\_bin, int \* blue\_bin){

//pixel coordinates

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

int tIndex = blockDim.x \* threadIdx.y + threadIdx.x;

if((tIndex %3) == 0){

unsigned int r = (unsigned int)(img[x][y].r/BIN\_WIDTH);

atomicAdd(& red\_bin[r], 1);

}

else if((tIndex %3) ==1){

unsigned int b = (unsigned int)(img[x][y].b/BIN\_WIDTH);

atomicAdd(& blue\_bin[b], 1);

}

else{

unsigned int g = (unsigned int)(img[x][y].g/BIN\_WIDTH);

atomicAdd(& green\_bin[g], 1);

}

}

**Method 3**

We create a local copy of histograms for each block and copy back the local copy to global memory at the end of calculation. This method has the advantage of taking less time as atomic operations are much faster on shared memory. In the last step we add all the partial histograms to calculate the final histogram.

\_\_global\_\_ void histogram\_localMem(int \* red\_bin, int \* green\_bin, int \* blue\_bin, int NUM\_BINS){

//pixel coordinates

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

int tIndex = blockDim.x \* threadIdx.y + threadIdx.x;

int g = blockIdx.x + blockIdx.y \* gridDim.x;

\_\_shared\_\_ unsigned int sMemRed[NUM\_BINS];

\_\_shared\_\_ unsigned int sMemBlue[NUM\_BINS];

\_\_shared\_\_ unsigned int sMemGreen[NUM\_BINS];

\_\_syncthreads();

unsigned int r = (unsigned int)(img[x][y].r/BIN\_WIDTH);

unsigned int b = (unsigned int)(img[x][y].b/BIN\_WIDTH);

unsigned int g = (unsigned int)(img[x][y].g/BIN\_WIDTH);

atomicAdd(& sMemRed[r], 1);

atomicAdd(& sMemBlue[b], 1);

atomicAdd(& sMemGreen[g], 1);

\_\_syncthreads();

int startPoint = blockDim.x\* blockIdx.y + blockIdx.x

for (int i = 3\* startPoint\* (NUM\_BINS-1); i < 3\* startPoint\* NUM\_BINS; i +=1) {

red\_bin[i] = sMemRed[i];

blue\_bin[i] = sMemBlue[i];

green\_bin[i] = sMemGreen[i];

}

}

cudaThreadSynchronize();

Once we have the partial histograms in the global memory we need to add them to the global histogram to have the final result.

//No of threads = NBlocks \* 3 \* B

\_\_global \_\_ void histogram\_accumulation(int \*red\_bin, int \* blue\_bin, int \* green\_bin, int \* final\_red\_bin, int \* final\_blue\_bin, int \* final\_green\_bin){

int tIndex = threadIdx.x + blockDim.x \* blockIdx.x;

int binNo = (tIndex%B)/3;

if((tIndex % 3) == 0){

readValue = red\_bin[binNo];

atomicAdd(&final\_red\_bin[binNo], readvalue);

}

else if ((tIndex % 3) == 1){

readValue = blue\_bin[binNo];

atomicAdd(&final\_blue\_bin[binNo], readvalue);

}

else{

readValue = green\_bin[binNo];

atomicAdd(&final\_green\_bin[binNo], readvalue);

}

}

**Question 3**

For each of the ways you came up with above, what is the complexity with regards to *N*, *P*, and *B*? As in all complexity type analysis, assume that both N and P are large and that *N >> P*. Please address three different cases: *B* = 24, *B* = 28, and *B* = 216 (i.e., *B* « *P*, *B* ~ *P*, and *B* » *P*).

Please report complexity for total computations, parallel computations, synchronization steps, and memory accesses (and any others you think would improve your understanding of the problem).

**Answer**

For this part assume certain more parameters as follows:

1. NBlocks = Number of blocks
2. SMP = Number of Streaming Multiprocessors
3. SP = Number of Streaming Processors per SMP such that SP\*SMP = P
4. NThreads = Total number of thread count
5. Total Computations

Method 1 & Method 2 only differ in the number of threads and not in the number of computations. So in Method 1 each thread will be doing 6 computations to find the bin number and then 3 atomic operations. Additionally we have one thread for each of the pixel. So the computation complexity will be

Computation Complexity = 6N + 3N (global memory atomic ops)

Now for method 3 we will have certain additional operations as we have to combine local histograms to forma the final histogram values.

Computation Complexity = 6N + 3N (shared memory atomic ops) + NBlocks\*3B

1. Parallel Computations

In Method 1, we do computations for all the pixels in parallel but number of Streaming Processors put a constraint on how many computations can be defined in parallel. Each thread is doing 3 Additions, 3 Divisions and 3 atomic operations on the global memory. So the total number of parallel computations is given by

Parallel Computations = 6P + 3P (global memory atomic operations)

In Method 2, each thread performs 1 Addition, 1 Division and 1 global memory atomic operation. SO the parallel computations are:

Parallel Computations = 2P + P (global memory atomic operations)

In Method 3, each of the thread performs 3 Additions, 3 Divisions and 3 share memory atomic operations in the histogram\_localMem karnel. Apart from these operations every thread in the histogram\_accumulation kernel performs 1 atomic addition in the global memory.

Parallel Computations Kernel 1 = 6P + 3P(shared memory atomic operations)

Parallel Computations Kernel 2 = 3\*NBlocks\*B (global memory shared operations)

1. Synchronization Steps

Method 1 and Method 2 requires no synchronization steps. Method 3 requires 2 synchronization steps within the histogram\_localMem kernel and one synchronization step between the two kernel calls.

1. Memory Access

**Question 4**

What is the memory complexity of the algorithms? Again, please consider the three different cases of *B* (or generalize the complexity equation). Memory complexity refers to the amount of memory required by your algorithms.

**Answer**

The memory complexity for first method is pretty straightforward as we have 3 global memory arrays for storing the histogram values.

So Memory Complexity = 3B

The second method also utilizes the same memory structure and as such it also has the same memory complexity of 3B.

For the third method we have local copies of histograms to reduce the time required in the computations and as a result the memory complexity increases.

In this method let us assume the following parameters:

1. NBlocks = Number of blocks
2. SMP = Number of Streaming Multiprocessors
3. NThreads = Number of Threads per block

The memory complexity of global memory in such a scenario is given by

Memory Complexity = (NBlocks+1)\*3\*B

In this the NBlocks\*3B refers to the local copies of the bins for each of the blocks and 3B for the final copy of the histogram.

For the local memory the memory complexity will be

Memory Complexity = (SMP)\*3B

This is due to the fact that at a certain instant there will be only so many local copies of the histogram as the number of Streaming Processors on which the code is getting executed.

Apart from this there will be utilization of the memory for saving the parts of images as well on which the threads are working at any moment.