CMPT431

Assignment 3

Rafiq Dandoo

Coding with Cuda

Coding with cuda felt pretty simple. It actually felt nicer than regular cpu multithreading because spawning the threads just involved creating __global__ function and calling it with function<<<blocks,threads>>>, then coding the actual gpu function was easy since you had simple access to Thread/Block Ids.

Naïve implementation

With the naïve implementation, the histogram calculation is done by 256 threads splitting the image up, then using a globally stored array and adding 1 atomically using cuda's atomicAdd() function.
This implementation is extremely simple to code and doesn't require much change from the sequential code, just changing how the for loop works and making the increment atomic since there's lots threads and there will be race conditions to increment the global histogram array.

The major downside to this, is having 256 threads compete for so few "buckets" as you could refer to it causes a lot of contention and slow down while using atomics. So this is where the major tiling change will come into play.

I did have to make the CDF calculation sequential because I couldn't see a way to make just due to the nature of how it's calculated using the sum of previous index values for the current index.

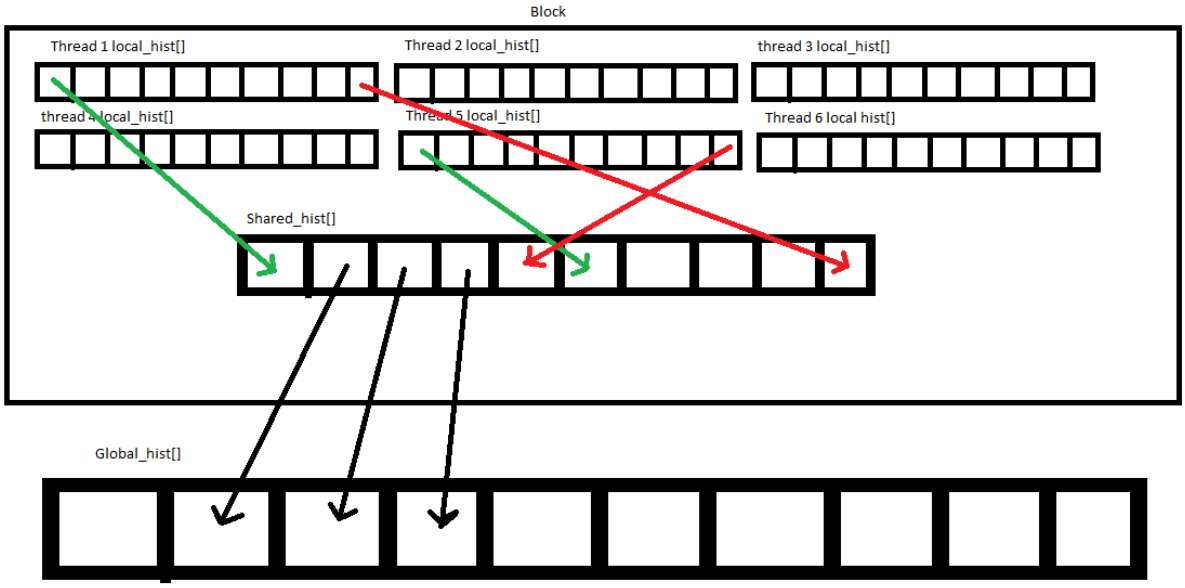
Tiled Implementation

So the main idea in a tiled implementation is data locality, so the main work being done where data locality can change is the histogram creation. The function does multiple atomicAdd() operations on a globally stored hist[] array, there are two problems here. The first one is that it's accessing global memory which is slow, over and over, the second is that it has to compete with 256 other threads for access to a certain bucket in the hist[] array.
So the way I solved this was by creating a local hist[] array in the register of each thread that they would sum up. Then for each N blocks, the 256 threads in that block would combine their local hist[] arrays into one that is in shared memory of the block. I also tried to do a little trick to reduce collisions between threads in the same block adding their value to the shared hist[]. I offset the bucket in which they start adding their values over by their threadId number. So thread 0 goes index 0 -> 256, thread 1 goes 1->256->0, thread 50 goes 50->256->49. I synch the

threads before they do this so hopefully this should reduce contention between threads, so it's not 256 threads trying to add to 0, then 1 and so on.

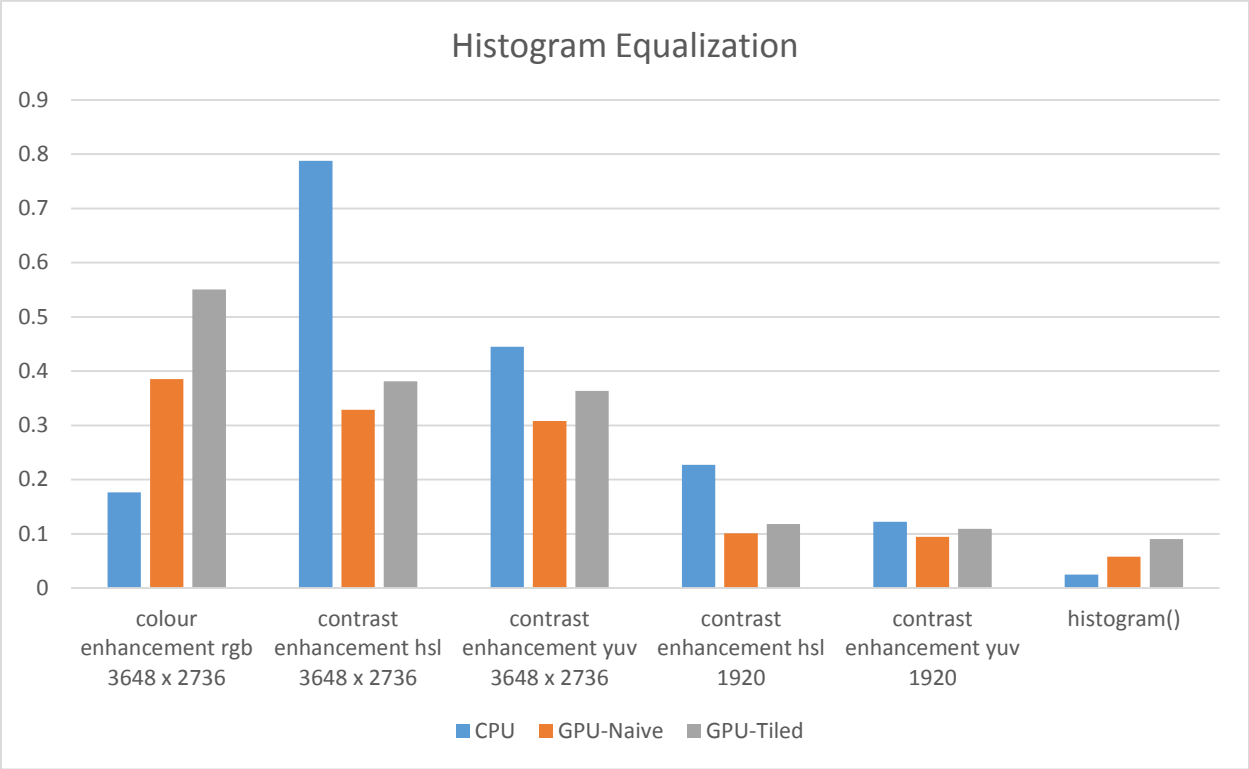
Then once the shared histogram is summed up I had each thread copy one bucket. Using it's thread ID as an index, to the globally stored hist[]. The idea was that this way you would only have N(the number of blocks) contentions between addition to the global bucket, if even that since some blocks may finish sooner or later.

Doing this should make the program run quicker since the histogram calculation is the core of the application since it is used by every version, gray, rgb, yuv, hsl, and in the last three it's used multiple times. And it has data that is being accessed multiple times from global memory.
The other functions like rgb2yuv, yuv2rgb, rgb2hsl, hsl2rgb, do have access to global memory, but in these cases you are only doing 1 memory access to each element in the array, and copying the array over into shared memory would take 1 access + another access when you use it, so it would always be slower.



This picture might help represent what I was doing with the tiling. The Green lines just show where the thread starts putting its values in, and the red one is where it ends. Then the values in the shared hist[] buckets are transferred to the global hist[] buckets by each thread In the block, since there are 256 buckets and 256 threads per block this works out so each thread just has to transfer 1 bucket over.

The Data



The data I collected is slightly different compared to what I was expecting. What I'm getting from the data is a few things.

The histogram function really doesn't benefit at all from trying to parallelize it, I'm not exactly sure why it's slower, the only reason I would think is because the use of atomics to prevent race condition slow it down. It's a bit surprising to me that it is faster for the sequential program to run through an array of 9 million size quicker than the parallel program can with the use of atomics to sum it up.

The above is proved by the RGB colour enhancement function which is essentially 3 histogram() functions each for the r g b values.

Since histogram clearly doesn't benefit, then what makes the HSL improve a lot? Well the conversion between RGB and HSL values requires quite a few floating point operations, so the while the sequential one has to do each index calculation one by one the gpu version is doing over a thousand indices in the array at a time.
The reason that HSL has a greater improvement over YUV is because YUV has fewer floating point operations so the cpu can speed through the indices pretty quickly.

What I learned from this is that GPUs have some very strong potential when used in the right situations much like any other programming tool and knowing when to use them is the key. The big one to me is remembering that just because a gpu has many cores it does not mean that just splitting up an array into segments will make it faster than a cpu going through it sequentially.

What seems to be the case is that for the GPU to outperform the cpus speed is that while going through those many iterations, there needs to be some complexity to compute which is where the GPU really shines and out preforms the CPU, as seen in the HSL example which requires a few floating point operations over 9 million indices, and is also proved from the lack of benefit of going over the same 9m items and just incrementing something as show in the histogram() function.

If I were to have redone the program in the future I would keep the histogram functions as sequential and kept the parallelizing of the RGB, HSL, YUV conversion functions.