COM4521 Assignment 2 Report

Design Considerations

The solution was to be developed as an extension to our submission for the first assignment. Each pixel of the image was represented as a Struct storing the values for Red, Green and Blue in the pixel. This means that we have an array of structures that has to be processed. This is bad for performance in Cuda so we have to rethink how to store this as a structure of arrays for faster processing.

Initial Implementation

To begin development I created a function to convert the structure of arrays into 3 1-dimensional arrays, one for each colour component. This should be enough to develop the kernel that averages the colour of the image.

This kernel would perform an atomic add to an int in memory that keeps the total for every r,g,b value to then calculate the average.

```
__global__ void get_image_averages(unsigned char* gpu_array_r, unsigned char*
gpu_array_g, unsigned char* gpu_array_b, int width, int height, int c) {
	// each thread loads one element from global to shared mem
	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	//sdata_r[tid] = gpu_array_r[i];
	//printf("thread %d, value %d", i, gpu_array_r[i]);

	atomicAdd(&gpu_total_r, gpu_array_r[i]);
	atomicAdd(&gpu_total_g, gpu_array_g[i]);
	atomicAdd(&gpu_total_b, gpu_array_b[i]);
}
```

The second kernel would average the colour of images inside a tile of size *C* and write back the colours for the pixels of the pixelate image

```
__global__ void cuda_image_pixelize(unsigned char* gpu_array_r, unsigned char*
gpu_array_g, unsigned char* gpu_array_b, unsigned char* gpu_out_r, unsigned
char* gpu_out_g, unsigned char* gpu_out_b, int width, int height, int tilesize)
{
	// map from threadIdx/BlockIdx to pixel position
	int x = threadIdx.x + blockIdx.x * blockDim.x;
	int y = threadIdx.y + blockIdx.y * blockDim.y;
	int output_offset = x + y * blockDim.x * gridDim.x;

	if ((output_offset == 0 || output_offset % tilesize == 0) && (y == 0 || y
% tilesize == 0)) {
		int avg_r = 0, avg_g = 0, avg_b = 0;
		for (int i = 0; i < tilesize; i++) {
			for (int j = 0; j < tilesize; j++) {
```

```
                        int index = output_offset + i + (j*height);
                        avg_r += gpu_array_r[index];
                        avg_g += gpu_array_g[index];
                        avg_b += gpu_array_b[index];
                  }
            }

            __syncthreads();
            for (int i = 0; i < tilesize; i++) {
                  for (int j = 0; j < tilesize; j++) {
                        int out_index = output_offset + i + (j * height);
                        //printf("out: %d \n",out_index);
                        gpu_out_r[out_index] = (unsigned char)(avg_r /
(float)(tilesize*tilesize));
                        gpu_out_g[out_index] = (unsigned char)(avg_g /
(float)(tilesize*tilesize));
                        gpu_out_b[out_index] = (unsigned char)(avg_b /
(float)(tilesize*tilesize));

                  }
            }
      }
}
```

This is the initial implementation of both kernels. Below we can find a table with different timings for these implementation.

Timings

I discovered after some experimentation that the optimum launch parameter for my kernel were as following. All of the timings for the report are taken utilising this parameters

```
      dim3 blocksPerGrid(width / 16, height / 16);
      dim3 threadsPerBlock(16,16);
```

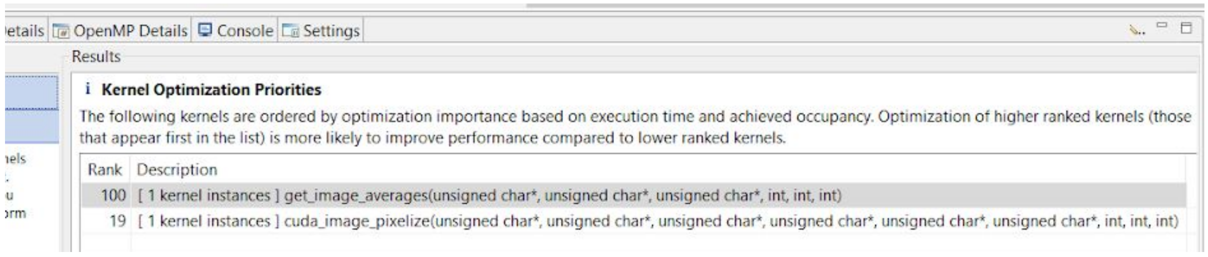
Tile Size **Image**	**2**	**8**	**16**	**32**	**64**
Sheffield 512x512	3ms	2ms	3ms	3ms	3ms
Dog 2048x2048	32ms	30ms	35ms	27ms	32ms

After obtaining all the timings for different possible scenarios I run the program through the Nvidia profiler to find out which parts of the code would benefit from some refactoring and code optimization.

Profiling Results

Small Images

The kernel optimization priorities returned this results for both the implemented kernels.

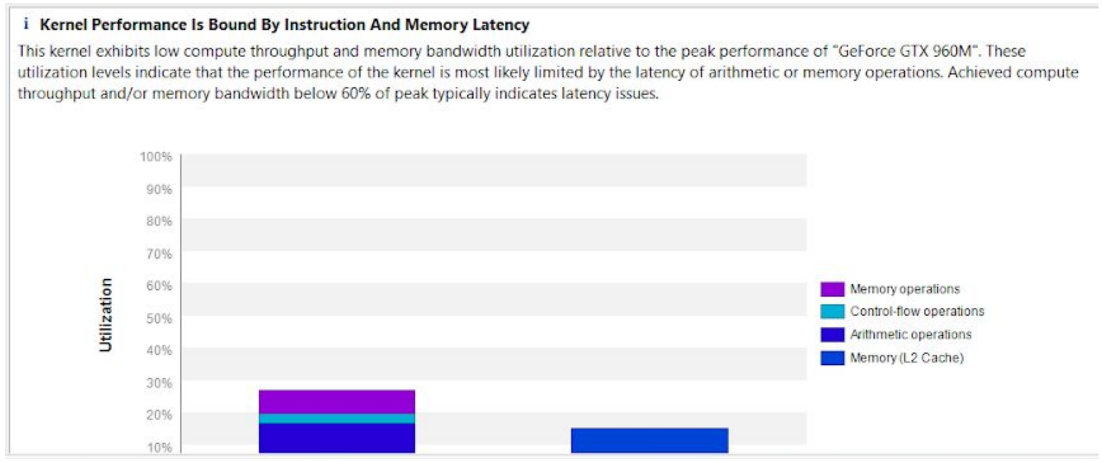


Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] get_image_averages(unsigned char*, unsigned char*, unsigned char*, int, int, int)
19	[1 kernel instances] cuda_image_pixelize(unsigned char*, unsigned char*, unsigned char*, unsigned char*, unsigned char*, unsigned char*, int, int, int)

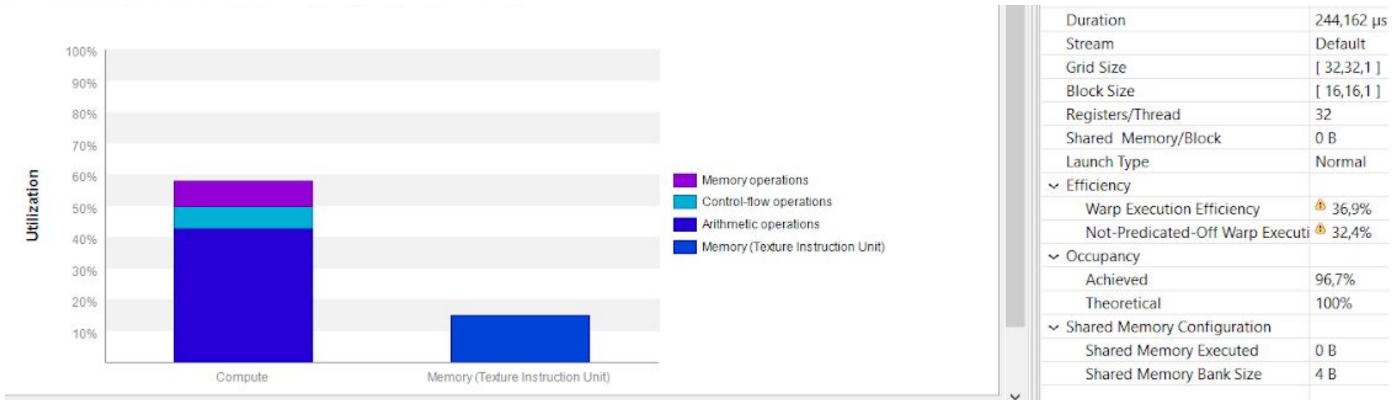
So it is clear from the results from the profiler that we will want to optimize the get_image_average kernel if we want to improve the execution time of the program. A quick kernel analysis on it shows that the kernel has a lot of latency and is probably cause by the AtomicAdd which makes the kernel modify a variable sequentially thus blocking the rest of executing kernels.



Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 960M". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

On the other hand, the cuda_image_pixelize kernel the kernel analysis results are as following. Better than the previous one but still bound by memory.



Duration	244,162 µs
Stream	Default
Grid Size	[32,32,1]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Efficiency	
Warp Execution Efficiency	36,9%
Not-Predicated-Off Warp Executi	32,4%
Occupancy	
Achieved	96,7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

This is probably due to me passing in three different arrays of r,g,b values which are separate in memory making the program run slower when having large images with rgb values being far away in memory.

Code Improvements

Assignment 1 Feedback

There were some test that were being failed by my program so based on the feedback from the first assignment I implemented fixes for these.
Firstly when calculating the image average I would store the total inside an *int,* this would overflow when large images were passed into the program so I converted this *int* into and *unsigned long long* which should be enough to make up for the overflow of the int.

Aside from this I reduces the compiler warning as much as possible, deleted some unused variables that went obsolete during development and cleaned up the code as much as possible.

Pixelate Filter

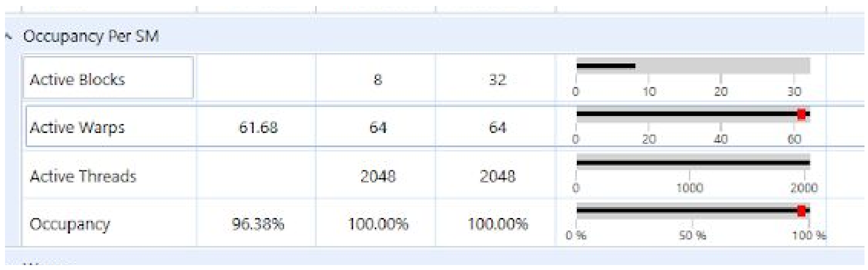
The initial implementation was based on the idea that an array of structures would have worse performance than a structure of arrays so I adapted my initial code to convert my 2D array of structures intro three 1D arrays, one for each colour component. This was good to get the logic implemented but in the end resulted in poor performance and a poor usage of memory.

I then changed this three 1D arrays into a 1D array of the vector uchar3, this would reduce the copies to memory which cause a large overhead and will keep the consequent values stored together in space.

Timings

Tile Size **Image**	**2**	**8**	**16**	**32**	**64**
Sheffield 512x512	3ms->1ms	2ms->1ms	3ms->2ms	3ms->2ms	3ms->2ms
Dog 2048x2048	32ms->13ms	30ms->18ms	35ms->13ms	27ms->21ms	32ms->24ms

The table above shows the improvements in time when changing the structure of the data. The improvement is significant for images of larger size when big arrays were passed into the Also the occupancy when up to 96%.



Occupancy Per SM

Active Blocks		8	32
Active Warps	61.68	64	64
Active Threads		2048	2048
Occupancy	96.38%	100.00%	100.00%

Image Average

I implemented reduction following the code from the lecture slides and produced the following code. This was to be executed inside the kernel that would average the image colors.

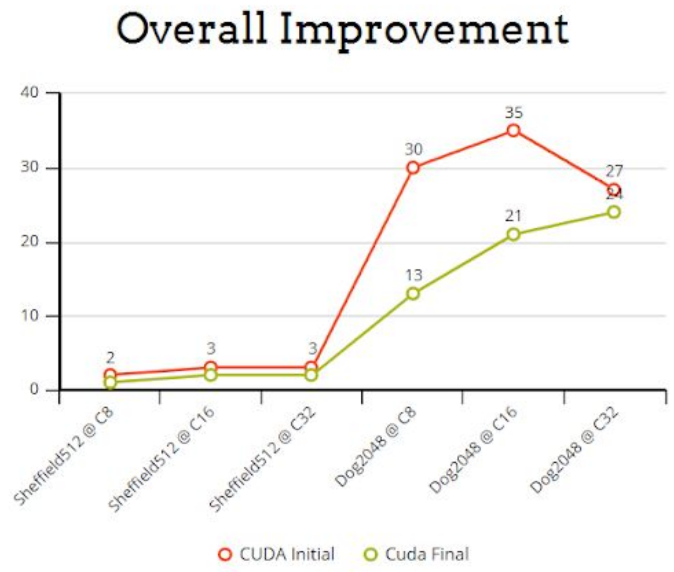
```
	extern __shared__ device_PPMPixel shared_data[];

	// each thread loads one element from global to shared mem
	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	shared_data[tid].red = (unsigned int) gpu_array_r[i];
	shared_data[tid].green = (unsigned int) gpu_array_g[i];
	shared_data[tid].blue = (unsigned int) gpu_array_b[i];
	__syncthreads();
	//printf("thread %d, value %d", i, gpu_array_r[i]);

	for(unsigned int stride = blockDim.x/2; stride >0; stride>>=1){
		if (threadIdx.x > stride) {
			shared_data[tid].red += shared_data[tid + stride].red;
			shared_data[tid].green += shared_data[tid +
stride].green;
			shared_data[tid].blue += shared_data[tid +
stride].blue;
		}
		__syncthreads();
	}

	if (threadIdx.x == 0) {
		atomicAdd(&gpu_total_r, shared_data[0].red);
		atomicAdd(&gpu_total_g, shared_data[0].green);
		atomicAdd(&gpu_total_b, shared_data[0].blue);
	}
```

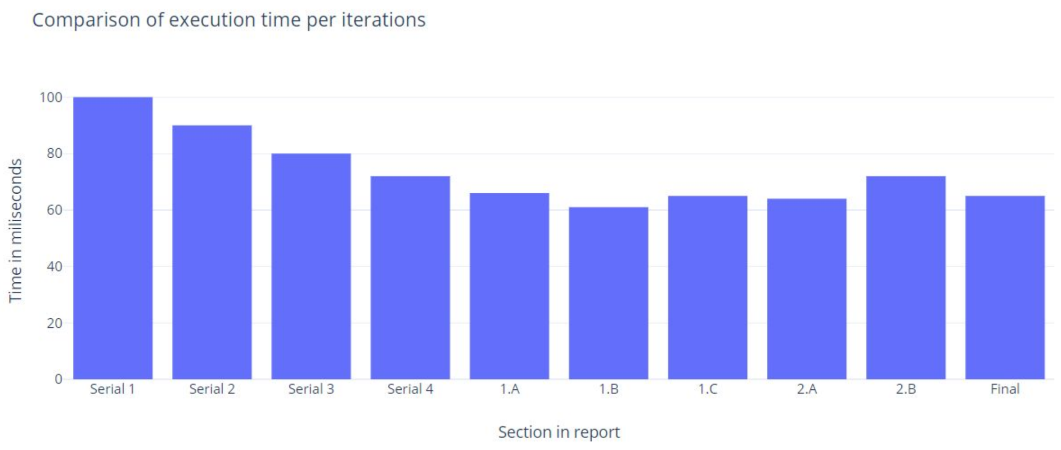
I found out that this was making the code run slower in average. I believe this could make a difference when very large images are passed into the filer but up to 2048x2048 the results have seemed to worsen by a 5-7 millisec.

I believe my initial implementation was a simplistic implementation that did just what it was meant to do in as little lines of code as possible so I am keeping the code for the initial implementation.

Conclusion And Results

If we compare the time recordings from this assignment and the previous one we can see that there is a great difference in the results. With the timings of the first assignment being of around 60 at best with value C=2 we can see that for larger values of C and larger images the timings are twice as fast at worst (Dog2048@16).





Even though the code improvements on the averaging function did not translate into a timing improvement, I'm happy that rethinking the data structures that were passed into the kernel did make a drastic change in the timings for larger images. Overall I am happy with the results of the program and consider it to be much better and correct for the scenarios in which it could be used.