Assignment_CUDA

180128022

May 2019

1 Introduction

CUDA (Compute Unified Device Architecture) is an extremely powerful parallel computing architecture and API (Application Programming Interface) which delivers the performance of NVIDIA's famous graphics processor technology to general purpose GPU Computing[1]. It allows software engineers to use a CUDA-enabled GPU (Graphics Processing Unit) for general purpose processing, termed GPGPU (General-Purpose computing on Graphics Processing Units). It is a software layer that allows the programmer to directly access the GPU's virtual instruction set or ISA (Instruction Set Architecture) and parallel computational elements, for the execution of compute kernels[2]. Below is the CUDA architecture provided by NVIDIA:

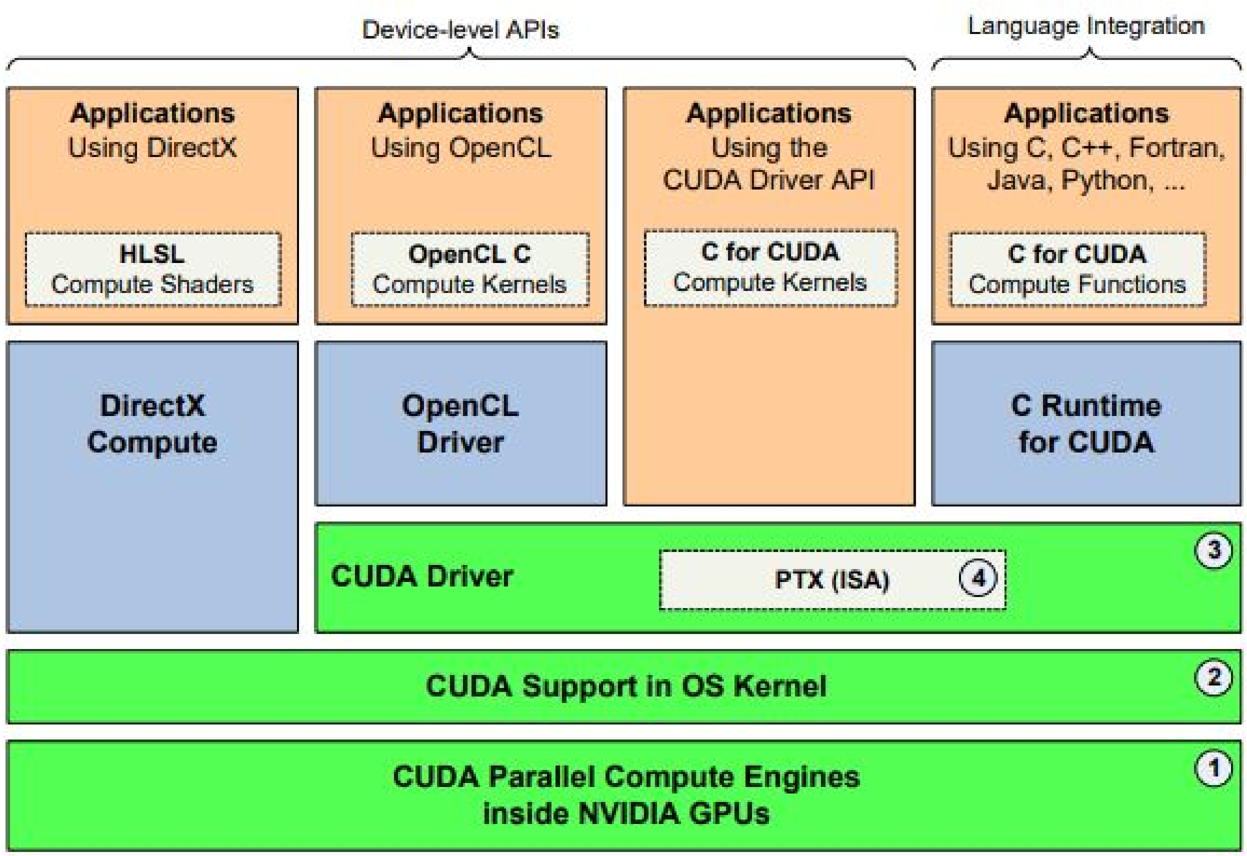


Figure 1: CUDA Architecture

2 Implementation

During the implementation for Assignment 1, there were quite a few issues specially with OPENMP and the use of Pragma Omp. Hence, this time around, the entire mosaic.c file had to be changed so that all the modes worked as per the requirement. First of all, the extension for mosaic.c file had to be changed to *.cu* so that GPU code could be added. After using the relevant imports, this time around, the approach of using function calls to respective functions was used. Screenshot below:

```c
// UID : 180128022

// imports
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <omp.h>

#define FAILURE 0
#define SUCCESS !FAILURE

#define USER_NAME "acp18rs"     //my user name

void print_help();


typedef enum MODE { CPU, OPENMP, CUDA, ALL } MODE; // type of modes

MODE execution_mode = CPU;

// declaration
unsigned char **red_img;
unsigned char **green_img;
unsigned char **blue_img;
char* str_cat(char *s1, char *s2);


// function call for command line process
int process_command_line(int argc, char *argv[], int *c, char *infile, char *outfile, char *PPM);
// function call for reading images
int in_img(char *infile, int c, int *width, int *height, char *header, char *outfile, char *PPM);
// function call for reading headers
int read_header(FILE *file, int c, int *width, int *height, char *header, char *outfile, char *PPM, char *format);
// function call to launch CPU mode
void CPU_launcher(int c, int width, int height, int *red_ave_host, int *green_ave_host, int *blue_ave_host);
// function call to launch OPENMP mode
void OPENMP_launcher(int c, int width, int height, int *red_ave_host, int *green_ave_host, int *blue_ave_host);
// function call to convert a 2D array to a 1D array
void converter_1D(unsigned char* red_CPU, unsigned char* green_CPU, unsigned char* blue_CPU, int width, int height);
// function call to launch CUDA mode
void CUDA_launcher(int c, int width, int height, unsigned char* red_CPU, unsigned char* green_CPU, unsigned char* blue_CPU, int *red_ave_host, int *green_ave_host, int *blue_ave_host);
// function call to convert 1D array back to 2D array
void converter_2D(unsigned char* red_CPU, unsigned char* green_CPU, unsigned char* blue_CPU, int width, int height);
// function call to check CUDA error if any
void errorCUDA(const char *msg);
// function call for writing images
int out_img(int width, int height, char *PPM, char *header, char *outfile);
```

Figure 2: Screenshot for Function Calls

New functions were defined that would read input image based on two conditions, PPM_Binary and PPM_Plain_Text, launch suitable mode of operation(CPU, OPENMP, CUDA and ALL) on the input image provided by the user to pixelate the same depending on the value of 'c' before writing the output to a *.ppm* file. Also, this time around the errors and exceptions have been gracefully handled as a part of the requirement. Below are two screenshots of code modules that prove the same:

```c
// checking if the value of c is greater than width or height of the image
if (c > *width || c > *height) {
    fprintf(stderr, "Error: The value of c has a higher value than either width or height of the original image. \n");
    return FAILURE;
}
```

Figure 3: Screenshot for Higher Input Image Dimension Handler

```
void errorCUDA(const char *msg) {
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err) {
        fprintf(stderr, "CUDA ERROR: %s: %s.\n", msg, cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}
```

Figure 4: Screenshot for CUDA Error Handler

The below code modules handles the values of mosaic cell size 'c' and input files which are incorrect (for example = 10 or an input file with the incorrect number of pixel values) and elegantly exists with a helpful error message:

```
cudaMemcpy(red_GPU, red_CPU, sizeof(unsigned char)*(width)*(height), cudaMemcpyHostToDevice);
cudaMemcpy(green_GPU, green_CPU, sizeof(unsigned char)*(width)*(height), cudaMemcpyHostToDevice);
cudaMemcpy(blue_GPU, blue_CPU, sizeof(unsigned char)*(width)*(height), cudaMemcpyHostToDevice);
errorCUDA("CUDA Error while copying memory from host to device!");
```

Figure 5: Screenshot for Incorrect Mosaic Cell Size Value Error Handler

```
// testing to check the existence of the file
if (file == NULL) {
    fprintf(stderr, "Error: Can not find the input file. \n");
    return FAILURE;
}
```

Figure 6: Screenshot for File-Not-Found Handler

2.1 CUDA-Specific Implementation

Initially, a 2D implementation was used, but due to the complexity of forcing the GPU to push the array values in to blocks and time required for execution, an **optimisation** had to be made, hence it had to be changed to a 1D implementation as no such complexity is present. The initial code for 2D implementation was something like this (it has not been included in the code):

Working Principle of previous Code: '__global__ void 2D_Kernel' function was called by the host to be executed on the GPU and x.Index and y.Index were assigned with the values (blockIdx.x * blockDim.x + threadIdx.x) and (blockIdx.y * blockDim.y + threadIdx.y) respectively. Initialising the float variables, it was ensured that the then-current thread was inside the boundary of the image. A loop was run that would update the corresponding red, green and blue values by summing them with a texture look-up in a given 2D sampler to ensure that out-of-range accesses were handled. The values were averaged to write to an output. In the CUDA_mode_launcher function, 2D memory was allocated to the device using cudaMalloc and data was copied from the host to device. The image was read using cudaBindTexture2D and the 2D Kernel was launched. The results were copied back to the CPU and both the Device and Host memories were freed.

Working Principle of current Code: Using __device__ a call from CPU is made to the GPU for the functions to be run on the GPU and __global__ is used for running a function on the device by calling from the device itself. As an **optimisation** method, shared memory is used since it is much faster than local and global memory. Thereafter, defining the conditions for appropriating the dimensions of the input image based on the 'c' value, atomicAdd is used to **to prevent**

Race conditions. To ensure that correct results are obtained when parallel threads cooperate, *_syncthreads()* is used. Then, the necessary conditions are taken care of, before synchronising the threads again. There is also a function to convert 2D arrays to 1D arrays and another one to convert the 1D arrays back to 2D arrays. A function called **CUDA_launcher** is used is defined to pixelate the input image on CUDA mode. In this functions, timers have been created, after which, memory has been allocated to the device, and simultaneously, exceptions have been handled. Then, data is copied to a symbol on the GPU by using *cudaMemcpyToSymbol* and the data is transferred from the host to the device. Here also, the code has been **optimised** to handle the exceptions. Thereby, defining the CUDA layout and execution method, the kernel is called from the host to the device. Finally, all the data is transferred back to the host and memory is freed before resetting the device.

In the main method, the placeholder for CUDA is filled with the program mode for the same. The **print_help()** function is updated to reflect the new CUDA option. Also, the **default mode** has been set to **CPU** in case of improper input mode.

```
// calling from GPU to be run on the GPU
__device__ int red_average_dev, green_average_dev, blue_average_dev;
// calling from CPU to be run on the GPU
__global__ void Kernel(unsigned char* red_GPU, unsigned char* green_GPU, unsigned char* blue_GPU, const int width, const int height, const int c) {

    // using shared memory as an optimisation method
    __shared__ int red_data, green_data, blue_data;
    // declaring variables
    unsigned int off;
    unsigned int i;
    int block_x = -1;
    int block_y = -1;
    int rem_x = 0;
    int rem_y = 0;
    int red_added = 0;
    int green_added = 0;
    int blue_added = 0;

    // conditions for appropriating the dimensions of the input image based on the 'c' value
    if (height % c != 0) {
        block_y = height / c;
        rem_y = height % c;
    }
    if (width % c != 0) {
        block_x = width / c;
        rem_x = width % c;
    }

    if (blockIdx.x != block_x || threadIdx.x < rem_x) {
        for (i = 0; i < blockDim.x; i++) {
            if (blockIdx.y == block_y && i >= rem_y)
                continue;

            off = (blockIdx.y * blockDim.x * width) + (blockIdx.x * blockDim.x + threadIdx.x) + width * i;
            red_added += red_GPU[off];
            green_added += green_GPU[off];
            blue_added += blue_GPU[off];
        }
        // using atomicAdd as an optimisation measure to prevent Race Conditions
        atomicAdd(&red_data, red_added);
        atomicAdd(&green_data, green_added);
        atomicAdd(&blue_data, blue_added);
    }

    // ensuring correct results when parallel threads cooperate
    __syncthreads();
    if (threadIdx.x == 0) {
        atomicAdd(&red_average_dev, red_data);
        atomicAdd(&green_average_dev, green_data);
        atomicAdd(&blue_average_dev, blue_data);
```

Figure 7: Screenshot for CUDA Implementation along with Optimisations

3 Results

c	Time in ms CPU	RGB Values CPU	Time in ms OPENMP	RGB Values OPENMP	Time in ms CUDA	RGB Values CUDA
2	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
4	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
8	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
16	0	red = 158 green = 162 blue = 170	0	red = 158 green = 162 blue = 170	0	red = 158 green = 162 blue = 170

Table 1: Table Containing Information for PPM_Binary on 16x16 Image Before Debugging

c	Time in ms CPU	RGB Values CPU	Time in ms OPENMP	RGB Values OPENMP	Time in ms CUDA	RGB Values CUDA
2	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
4	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
8	0	red = 158 green = 162 blue = 170	0	red = 157 green = 161 blue = 170	0	red = 157 green = 161 blue = 170
16	0	red = 158 green = 162 blue = 170	0	red = 158 green = 162 blue = 170	0	red = 158 green = 162 blue = 170

Table 2: Table Containing Information for PPM_Plain_Text on 16x16 Image Before Debugging

c	Time in ms CPU	RGB Values CPU	Time in ms OPENMP	RGB Values OPENMP	Time in ms CUDA	RGB Values CUDA
2	73	red = 125 green = 126 blue = 111	69	red = 124 green = 126 blue = 111	86	red = 161 green = 163 blue = 144
4	65	red = 125 green = 126 blue = 111	65	red = 124 green = 125 blue = 111	36	red = 132 green = 133 blue = 117
8	63	red = 125 green = 126 blue = 111	64	red = 124 green = 125 blue = 111	17	red = 126 green = 127 blue = 112
16	64	red = 125 green = 126 blue = 111	63	red = 124 green = 125 blue = 111	10	red = 125 green = 126 blue = 111
32	65	red = 125 green = 126 blue = 111	64	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
64	63	red = 125 green = 126 blue = 111	63	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
128	64	red = 125 green = 126 blue = 111	64	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
256	65	red = 125 green = 126 blue = 111	65	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
512	63	red = 125 green = 126 blue = 111	63	red = 124 green = 125 blue = 111	8	red = 124 green = 125 blue = 111
1024	63	red = 125 green = 126 blue = 111	62	red = 124 green = 126 blue = 111	12	red = 124 green = 126 blue = 111

Table 3: Table Containing Information for PPM_Binary on 2048x2048 Image Before Debugging

c	Time in ms CPU	RGB Values CPU	Time in ms OPENMP	RGB Values OPENMP	Time in ms CUDA	RGB Values CUDA
2	45	red = 125 green = 126 blue = 111	43	red = 124 green = 126 blue = 111	11	red = 166 green = 167 blue = 148
4	63	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	6	red = 133 green = 134 blue = 118
8	41	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	5	red = 126 green = 127 blue = 112
16	67	red = 125 green = 126 blue = 111	40	red = 124 green = 125 blue = 111	5	red = 125 green = 126 blue = 111
32	40	red = 125 green = 126 blue = 111	42	red = 124 green = 125 blue = 111	5	red = 124 green = 125 blue = 111
64	41	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
128	41	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
256	61	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	5	red = 124 green = 125 blue = 111
512	40	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	5	red = 124 green = 125 blue = 111
1024	40	red = 125 green = 126 blue = 111	40	red = 124 green = 126 blue = 111	5	red = 124 green = 126 blue = 111

Table 4: Table Containing Information for PPM_Binary on 2048x2048 Image After Debugging

c	Time in ms CPU	RGB Values CPU	Time in ms OPENMP	RGB Values OPENMP	Time in ms CUDA	RGB Values CUDA
2	72	red = 125 green = 126 blue = 111	68	red = 124 green = 126 blue = 111	84	red = 166 green = 167 blue = 148
4	65	red = 125 green = 126 blue = 111	64	red = 124 green = 125 blue = 111	35	red = 133 green = 134 blue = 118
8	63	red = 125 green = 126 blue = 111	62	red = 124 green = 125 blue = 111	16	red = 126 green = 127 blue = 112
16	63	red = 125 green = 126 blue = 111	62	red = 124 green = 125 blue = 111	9	red = 125 green = 126 blue = 111
32	65	red = 125 green = 126 blue = 111	64	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
64	62	red = 125 green = 126 blue = 111	62	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
128	63	red = 125 green = 126 blue = 111	61	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
256	63	red = 125 green = 126 blue = 111	63	red = 124 green = 125 blue = 111	7	red = 124 green = 125 blue = 111
512	62	red = 125 green = 126 blue = 111	62	red = 124 green = 125 blue = 111	8	red = 124 green = 125 blue = 111
1024	62	red = 125 green = 126 blue = 111	62	red = 124 green = 126 blue = 111	12	red = 124 green = 126 blue = 111

Table 5: Table Containing Information for PPM_Plain_Text on 2048x2048 Image Before Debugging

c	**Time in ms CPU**	**RGB Values CPU**	**Time in ms OPENMP**	**RGB Values OPENMP**	**Time in ms CUDA**	**RGB Values CUDA**
2	43	red = 125 green = 126 blue = 111	42	red = 124 green = 126 blue = 111	11	red = 166 green = 167 blue = 148
4	41	red = 125 green = 126 blue = 111	40	red = 124 green = 125 blue = 111	6	red = 133 green = 134 blue = 118
8	41	red = 125 green = 126 blue = 111	40	red = 124 green = 125 blue = 111	5	red = 126 green = 127 blue = 112
16	40	red = 125 green = 126 blue = 111	40	red = 124 green = 125 blue = 111	4	red = 125 green = 126 blue = 111
32	41	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
64	41	red = 125 green = 126 blue = 111	42	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
128	40	red = 125 green = 126 blue = 111	40	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
256	41	red = 125 green = 126 blue = 111	43	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
512	65	red = 125 green = 126 blue = 111	41	red = 124 green = 125 blue = 111	4	red = 124 green = 125 blue = 111
1024	40	red = 125 green = 126 blue = 111	65	red = 124 green = 126 blue = 111	5	red = 124 green = 126 blue = 111

Table 6: Table Containing Information for PPM_Plain_Text on 2048x2048 Image After Debugging

Figure 8: Pixelated 2048x2048 Image with c = 64

Figure 9: Pixelated 16x16 Image with c = 2

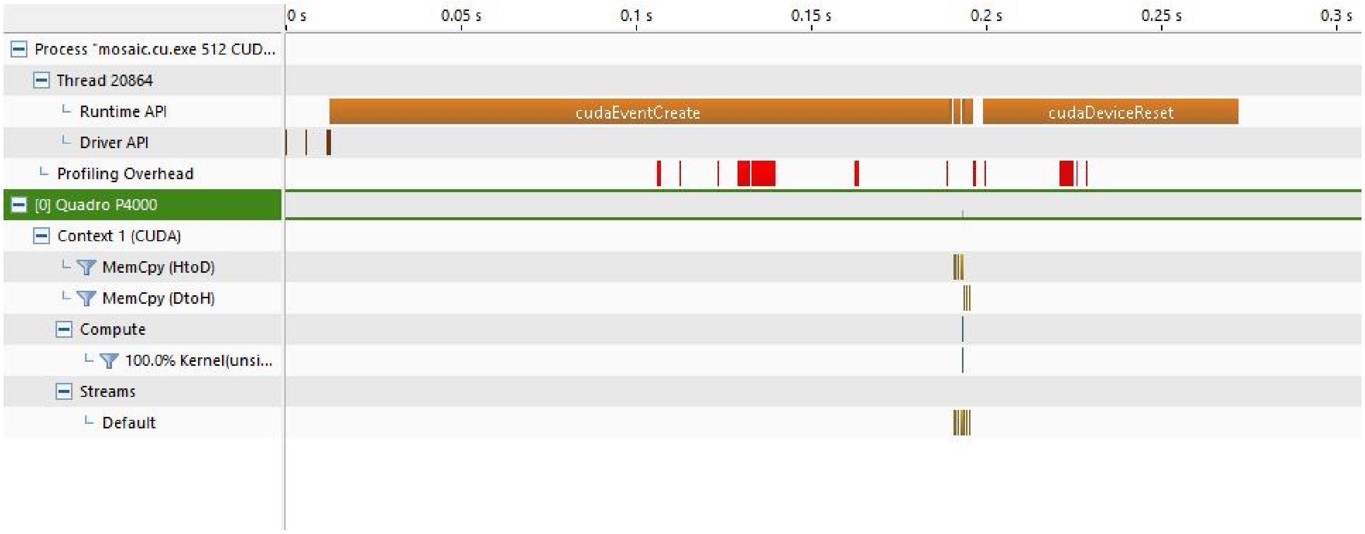
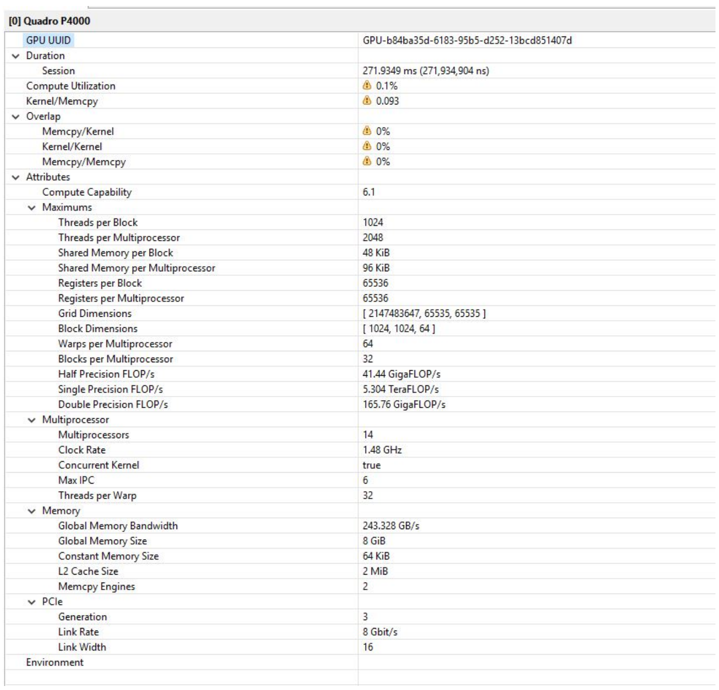


Figure 10: Profiler Image 1



[0] Quadro P4000	
GPU UUID	GPU-b84ba35d-6183-95b5-d252-13bcd851407d
Duration	
Session	271.9349 ms (271,934,904 ns)
Compute Utilization	0.1%
Kernel/Memcpy	0.093
Overlap	
Memcpy/Kernel	0%
Kernel/Kernel	0%
Memcpy/Memcpy	0%
Attributes	
Compute Capability	6.1
Maximums	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	96 KiB
Registers per Block	65536
Registers per Multiprocessor	65536
Grid Dimensions	[2147483647, 65535, 65535]
Block Dimensions	[1024, 1024, 64]
Warps per Multiprocessor	64
Blocks per Multiprocessor	32
Half Precision FLOP/s	41.44 GigaFLOP/s
Single Precision FLOP/s	5.304 TeraFLOP/s
Double Precision FLOP/s	165.76 GigaFLOP/s
Multiprocessor	
Multiprocessors	14
Clock Rate	1.48 GHz
Concurrent Kernel	true
Max IPC	6
Threads per Warp	32
Memory	
Global Memory Bandwidth	243.328 GB/s
Global Memory Size	8 GiB
Constant Memory Size	64 KiB
L2 Cache Size	2 MiB
Memcpy Engines	2
PCIe	
Generation	3
Link Rate	8 Gbit/s
Link Width	16
Environment	

Figure 11: Profiler Image 2

4 Discussion

1. Table 1 shows information of execution times and individual values for a **16x16 PPM_Binary** image before changing the Configuration mode to 'Kernel' under Debugging. It is seen that all the modes take no time to get executed since its size is extremely small. It remains the same after changing the Configuration to Kernel.
2. Similarly, Table 2 shows information of execution times and individual values for a **16x16 PPM_Plain_Text** image before changing the Configuration mode to 'Kernel' under Debugging. It is seen that all the modes take no time to get executed since its size is extremely small. It remains the same after changing the Configuration to Kernel just like the previous mode.
3. Table 3 shows information of execution times and individual values for a **2048x2048 PPM_Binary** image before changing the Configuration mode to 'Kernel' under Debugging. It is seen that for higher values of mosaic cell size, 'c', time taken to execute CUDA mode is much less than any other mode, which signifies **parallelisation**.
4. Table 4 shows information of execution times and individual values for a **2048x2048 PPM_Binary** image after changing the Configuration mode to 'Kernel' under Debugging. It is seen that for higher values of 'c', time taken to execute CUDA mode is way too less than before, which signifies **parallelisation**.
5. Table 5 shows information of execution times and individual values for a **2048x2048 PPM_Plain_Text** image before changing the Configuration mode to 'Kernel' under Debugging. It is seen that for higher 'c', time taken to execute CUDA mode is much less than any other mode, which signifies **parallelisation**.
6. Table 6 shows information of execution times and individual values for a **2048x2048 PPM_Plain_Text** image after changing the Configuration mode to 'Kernel' under Debugging. It is seen that for higher values of 'c', time taken to execute CUDA mode is way too less than before, which signifies **parallelisation**.
7. It is seen that the execution time for CUDA on Plain Text files is a slightly less than that for Binary files.

5 Conclusion

- **Parallelisation:** Parallelisation has been successfully achieved as seen from the time required for CUDA execution.
- **Structures of Arrays:** Different methods have been used to layout the data in GPU memory to ensure coalesced access patterns.
- **GPU Memory Caches:** Shared memory has been used to lower latency.
- GPU Optimisations: The various optimisations have been discussed above from using GPU Memory Caches to avoiding Race Condition.
- **Other Optimisations:** Other optimisations include use of 'char' this time around compared to using 'int' on the previous assignment, as much more memory is required for 'int' (16 bits) compared to 'char' (8 bits).
- **Description of Technique:** The technique used to implement the code has been discussed.
- **Profiling:** The screenshot of the NVIDIA CUDA Profiler is provided.
- **Discussion about Performance:** Many interesting aspects of CUDA are evident from the tables above, which give a description of its performance measures.

References

[1] Developer.download.nvidia.com. (2019). [online] Available at: http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf.

[2] En.wikipedia.org. (2019). CUDA. [online] Available at: https://en.wikipedia.org/wiki/CUDA.

[3] tack Overflow. (2019). Newest 'cuda' Questions. [online] Available at: https://stackoverflow.com/questions/tagged/cuda.

[4] eveloper.download.nvidia.com. (2019). [online] Available at: http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf.