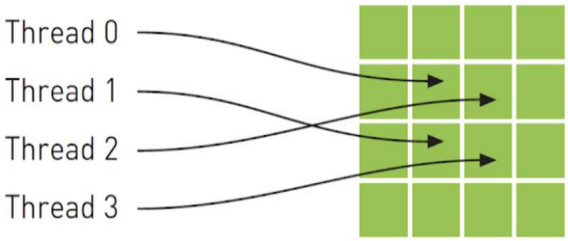
GPU Assignment: Part 2 Report

Design Consideration

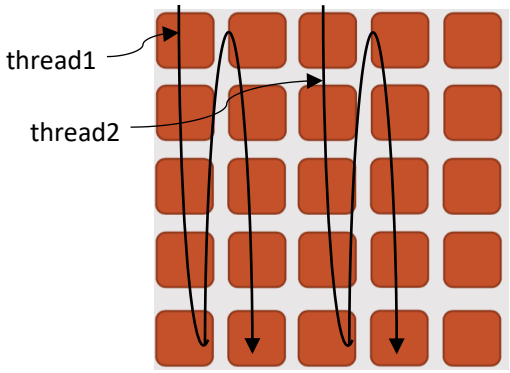
At the beginning, I try to think about how to address image more easily and intuitively, and 2D texture memory is my first chose. Compared with constant memory, texture memory is another variety of read-only memory which can improve performance and reduce memory traffic when we read that have certain access patterns. In other words, it could be originally designed for the graphics applications, and also be used quite effectively in the image processing. More specifically, for computing the RGB values, texture memory roughly implies that a thread is likely to read and deal with a pixel each time from an address near an address that nearby threads read.



cut from: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

As can be seen from the graph above, the implementation approach could be similar to the CPU mode, and I can derive correct index position directly through simple nest for loop. Arithmetically, these four addresses shown are not consecutive, so that they would not be cached together in a typical CPU caching scheme. However, GPU texture caches seems to be designed to accelerate access patterns, and theoretically, the performance may increase in this case when using texture memory instead of global memory. The execution results will be shown in conclusion and results section, the fact is totally out of blue.

On the other hands, I also implement a basic GPU kernel function after transferring the 2-dimisional unsigned char array into 1-dimension that can make much easier to find out the index only using x axis of a thread index and does not care about y or z. In addition, an atomic function is essential. It reads a value at some address in global or shared memory, adds a number to it, and writes the results back to the same address. An advantage of the function is that it is guaranteed to be performed without interference from other threads, but reduces the performance at same time. Regarding to performance concern, adding up all the values within a single block for one mosaic cell as possible is better than that across several blocks which means that more global memory and atomicAdd function we have to be used. In order to achieve that, I parallelize that one thread in a block addresses not only a pixel but also a column because it could be quite fixable for different size of c we assigned. For instance, based on the constraint of the maximum number of threads in a certain block, if the c size is smaller than 1024, each thread could deal with one column whose number of pixel is same to the c value. It can be seen clearly from below graph that there are two or three columns should be summed up for each thread in a block when the c size is larger than 1024.



It is noticeable that, I can just use a simple variable to compute the sum within each thread and then sum them up through a shared memory between different threads instead of calling atomicAdd frequently in the nest loop. Their performance will also be compared in optimization and performance section.

Parallel Implementation

- **Fix bugs from assignment 1**

  Before implementing CUDA function, there are several issues from the feedback I need to fix. First, the input handling should be thought more rigorous and that could cause the program to crash or terminate unusually. For example, the default of output format could be set as BINARY and its option -f is not necessary which means that the number of arguments is either seven or nine. In addition, if the input is not kind of ppm file, it should not be allowed to continue. In particular, the mosaic cell c is limited to be powers of 2, and its value is also less than the width or height of image and larger than zero. Second, the variables, image_r, image_g, and image_b, which store the RGB values respectively , should be represented as unsigned char instead of integer array into because unsigned char can only access between 0 and 255 whose range is exactly suitable for ppm image. After that, I have to change the data type of those sum variables from int to unsigned long long int. The int only permits values from -2147483648 to 2147483647, and it could result in overflow problem when the size of input image is over ten thoudsands, that is, the each sum among RGB could reach 10000x10000x255. Compared with that, it is enough that the unsigned long long int can represent positive integer over 19 digits. Last one, some complier warning messages including unsafe data type transformation must be fixed. Due to the fact that the warning message like below could happen as lager data type (8 bytes) for summerising all image values transfers to small one (1 bytes) for representing RGB values. It could be solved using brackets to the computated results before casting to small data type forcely.

```
warning C4244: '=': conversion from 'unsigned __int64' to 'unsigned char', possible loss of data
```

  Furthermore, it is interested in that I have to assign the modifier “%hhu” as the second parameter of fscanf function rather than “%u” to fix compiler warning although both of them gain the same values.

- **Implement with texture 2D memory**

  I try to use texture 2D memory to do the pixelate filter, and it is quite rough and straightforward version because I do not optimize them after moving on to another approach. For the implementation of my first version, cudaMallocPitch, cudaMemcpy2D, and cudaBindTexture2D are the most special and different with traditional functions. First, before utilising the texture memory, we need to declare its template type like *texture<unsigned char, cudaTextureType2D> texRef* which is just supported on global variables right now. After that, for allocations of 2D arrays, it is recommended to perform pitch allocations using cudaMallocPitch function due to pitch alignment restrictions in the hardware. Before calling to kernel function, those device memory which already allocated should be combined and linked to texture reference through cudaBindTexture2D. Because of this step, we can easily access the texture contents in kernel function by tex2D provided from CUDA. In terms of reading RGB values of image, it is quite similar to CPU mode adding the offset i and j up x and y index to gain the nearby values with a square. This is also the biggest difference from 1D method shown on below.

  ```
  output_r += tex2D(texData_r, xIndex + i, yIndex + j);
  ```

  In particular, the CUDA texture provides several address modes: clamp, border, warp and mirror. In here, I use the cudaAddressModeBorder which could automatically handle out-off-range access.

- **Implement for 1D array**

  Because of default 2-dimision image arrays used by the CPU and OPENMP modes, I definitely have to write two functions for transferring the 2D array to 1D and turn back to 2D after executing the kernel code. In this section, I will focus on how to implement those code on the device and explain how to find out the correct offset for 1D array though the built-in library. It could be divided into three parts in the kernel.

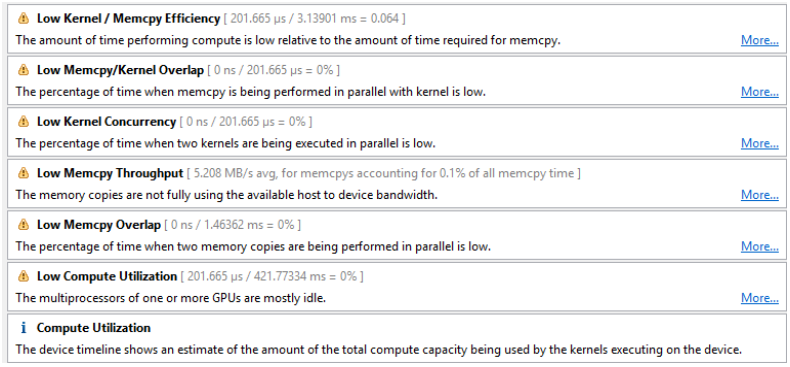
  The first part is that summing up all the image values. For 1D array, I could use the blockIdx and threadIdx to assign which index is my target. The *blockIdx.y*width*c* represents how many rows it may pass totally before the target row we are looking for. The reason why defining the number of block through x and y axis is that it is much clear and direct to assign the height rather than computed just by x index. The *blockIdx.x*c+threadIdx.x* means how many pixels it may pass at the target row. In kernel function, threadIdx.x could traverse all the threads within a block in order automatically. For *j*1024* and *width*i*, they can choose which index on a certain row and column respectively within a single thread is the target. For the next part, all the values stored via shared memory need to be divided from the number of pixels in a block. There are different numbers at boundary block index. Particularly, before moving on to next part, *__syncthreads* function, kind of thread barrier waiting for the others in that block, is crucial. Finally, all the RGB values of image should be replaced by the average value based on which block they located. The way for looking for index is completely similar to the first part.

Optimisation and Performance

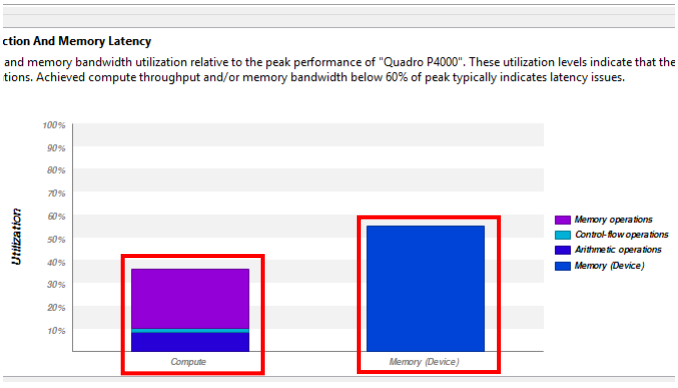
In this section, I will analysis the performance of my program from two parts: execution time of two different implementation approached (texture 2D and 1D array) and computation and memory utilization focused on 1-dimension method using NVIDIA Visual Profiler.

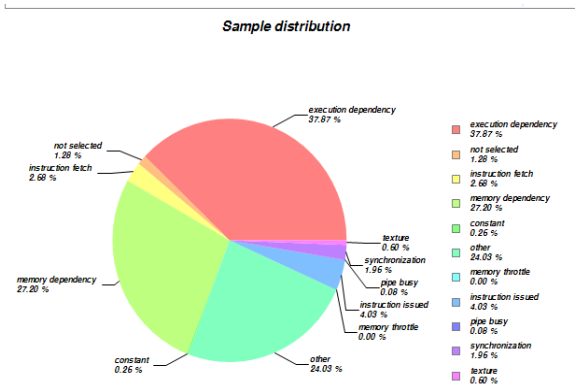
First of all, I test the execution time of textures 2D, it could take about **five hundred milliseconds** for 2048x2048 pixels image, and it also takes more than hundred milliseconds for small images. It seems to be a little bit wired because texture memory should deal with specific 2D graph basically as I mentioned above. Even though I have not optimized them too much, it should not have such big gap of time compared with 1D. After asking the assistants who are also confused about that, and it is recommended that the 1D way might be gain better performance for this assignment. I guess that, in this case, the pointer must jump over the c size to the next mosaic cell for computing average which could be not traditional image processing pixel by pixel, and the times of accessing global memory may increase.

After that, I change to do the work though 1D approach and try to optimize them. For my first profiling results, there are several issues I found. As can be seen from the graph below, I think it is quite hard to fix all of them because my kernel function should be fast enough when all of threads in a single block are activate. In next section, the execution time will be shown. Moreover, it is evitable for adding **__syncthreads** function for the sake of race condition issues and ensuring correct computation.

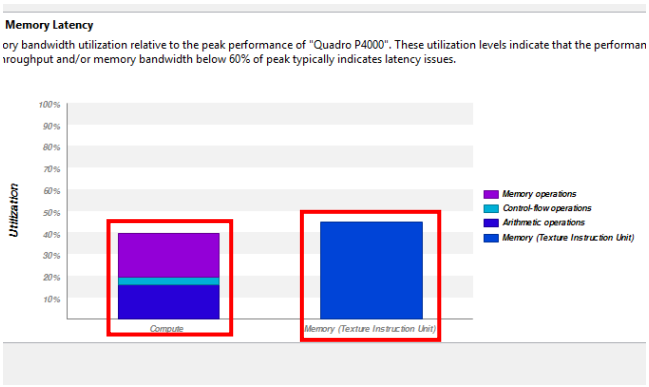


However, there are many benchmarks to be determined the program such as compute and memory utilization. It is clearly that both utilization of compute and memory in the device are lower than sixty percentage which means that there are many resources being wasted and not paralleled enough.

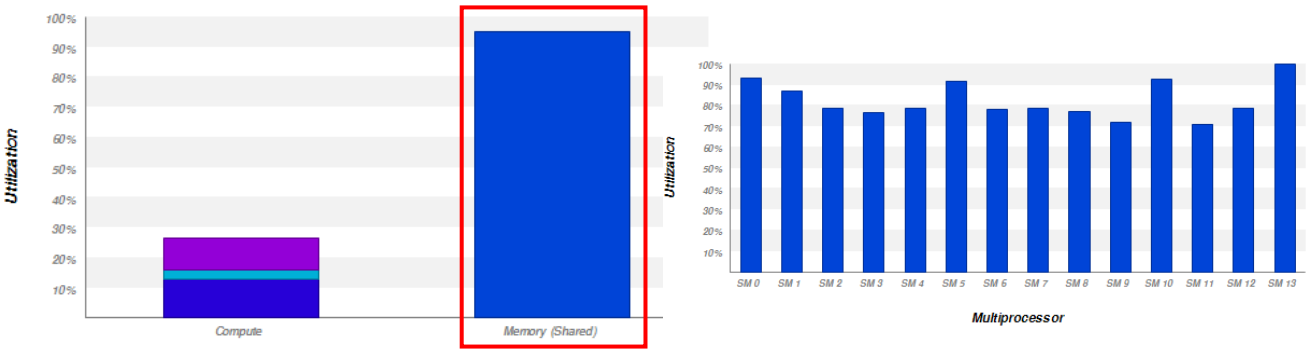




Next step, I just use some local variables in the kernel function to sum up the values derived from the same threads, and then summing up them using shared memory within a single block in order to reduce the number of atomicAdd function. Although the execution can decrease slightly around 2 or 3 millisecond and compute utilization also increases, the memory utilization even drops dramatically to less than 50%.



According to the messages given from profiler results, I transfer three unsigned char data type into a single vector types uchar3 derived from the basic integer or floating-point types because the help information shows that it could be better for minimising data transfer between the host and the device. I think that uchar3 dimension may be allocated consecutive memory spaces instead of three distinct spaces. The result of graph illustrates that the memory should be completely utilised in the kernel which is up to over ninety percent and all the multiprocessor also work efficiently.

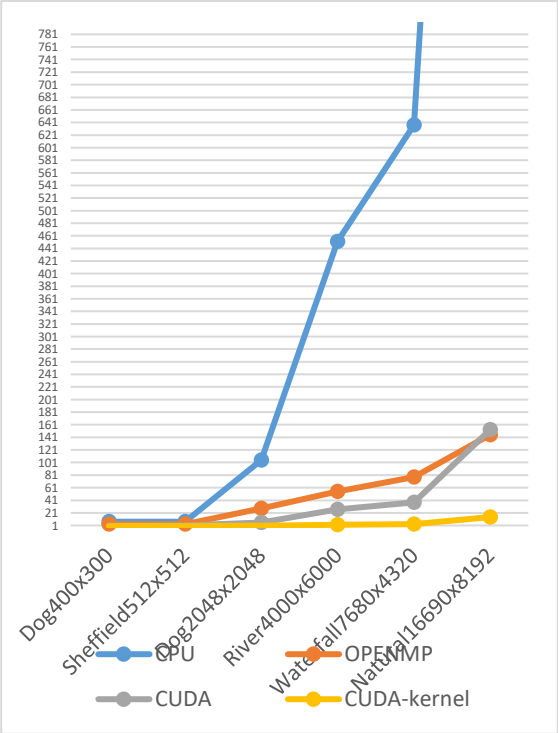


Regarding to the lower compute utilization, it seems to be a normal situation due to the fact that there are few overlap for computing and all the computation in different parts mentioned above has to wait for the results calculated from previous one.

Conclusion and Results

In this section, some ppm images downloaded from the Internet with different sizes are tested via different execution modes. As can be seen from the line chart below, the execution time of CUDA mode is exactly quicker than that of others no matter what sizes images I input. It is noticeable that it is really fast for only examining the execution time of the kernel function, that is, most time could be taken on memory allocation and memory copy rather than the code in kernel. In addition, the kernel function only takes near 15 milliseconds on larger image

which is the same time consuming as someone who optimizes his code through the reduction or shuffle methods. For me, they are not necessary.



Milliseconds

	CPU	OPENMP	CUDA	CUDA-kernel
Dog400x300	7	3	0.667	0.044
Sheffield512x512	7	3	0.905	0.06
Dog2048x2048	105	28	5.561	0.618
River4000x6000	452	55	26.285	2.235
Waterfall7680x4320	637	78	37.808	3.381
Natural16690x8192	2570	145	152.953	14.303