B Tech Project

First Stage Report on

GPU Based Image Processing

Submitted by

Sumit Shekhar

(05007028)

Under the guidance of

Prof Subhasis Chaudhari

1. Introduction

1.1 Graphic Processor Units

A graphic processor unit is simply a processor attached to the graphics card used in video games, play stations and computers. The way they are different from the CPUs, the central processing units, are that they are *massively threaded* and *parallel* in their operations. This is because of the nature of their work – they are used for fast rendering; same operation is carried out for each of the pixels in the image. Thus, they have *more* transistors devoted to *data processing* rather than flow control and data caching.

Today graphic processor units have outdated their primary purpose. They are being used and promoted for scientific calculations all over the world by the name of GPGPUs or general purpose GPUs; engineers are achieving several times speed-up by running their programs on GPUs. Applications fields are many: image processing, general signal processing, physics simulations, computational biology, etc, etc.

1.2 Problem Statement and Motivation

Graphic Processor Units can speed-up the computation manifolds over the traditional CPU implementation. Image processing, being inherently parallel, can be implemented quite effectively on a GPU. Thus, many applications which are otherwise run slowly can be fastened up, and can be put to useful real-time applications. This was the motivation behind my final year project.

NVIDIA *CUDA* is a parallel programming model and software, which has developed specifically to address the problems of efficiently program the GPU as well as be compatible with a wide variety of GPU cores available in the market. Further, being an extension to the standard C language, it presents a low-learning curve for the programmers, as well giving them flexibility to put in their creativity in the parallel codes they write.

My task was to implement object-tracking algorithms using CUDA and optimize them. As a part of the first stage, I implemented *Bilateral Filtering* method on GPU. Traditionally, brute force bilateral filters take a long time to run because (i) they cannot be implemented using FFT algorithms as the calculation involves both spatial and range filtering and, (ii) they are not separable, hence takes $O(n^2)$ computations. Using GPU, I found them to be running much faster than even on a high-end CPU.

2. CUDA Programming Model

2.1 Execution

CUDA extends the standard C language to make it applicable for parallel programming. Its various features include:

- C functions are implemented on GPU device using *kernels,* which are executed in parallel in several *CUDA threads,* as opposed to C functions which are executed only once.
- The *kernels* are defined using **`__global__`** identifier, which is again an extension of CUDA. The kernel function is called using a special **<<<…>>>** syntax, which specifies the number of threads in which the kernel has to execute.
- The **<<<…>>>** allows to determine the organization of threads and how they are executed. A typical syntax for calling a *kernel* function is shown as below:

```
funcAdd<<<Grid, Block, 1>>>(A, B, C)
```

- This illustrates how a general function is executed on the GPU in CUDA. Block variable defines a block of threads, which can be one-dimensional, two-dimensional or three-dimensional. Each thread in the block is identified by a 3-component vector called **`threadIdx`,** whose x, y and z components respectively gives a unique index to each thread in a block.
- Similarly, **`Grid`** defines the layout of the blocks, which can be either one-dimensional or two-dimensional. The blocks are also identified by a vector called **`blockIdx`**.
- Each block has a limit on the maximum number of threads it can contain, which is determined by the architecture of the unit. These threads can be synchronized with each other using **`__syncthreads()`** functions and can also be made to access the memory in synchronization. The grid and blocks can be shown by following diagram:

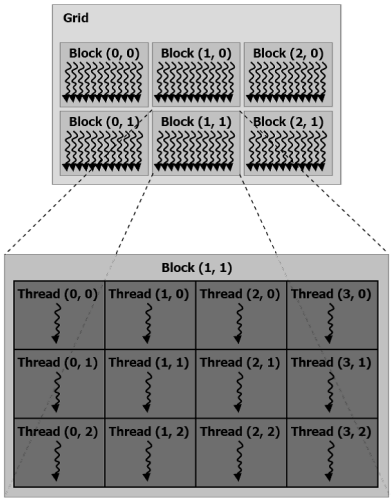

Figure 1: Grid and Thread Blocks

2.2 Memory Hierarchy

There are multiple memory spaces which a CUDA thread can use to access data. The different kinds available are shown below in the figure:

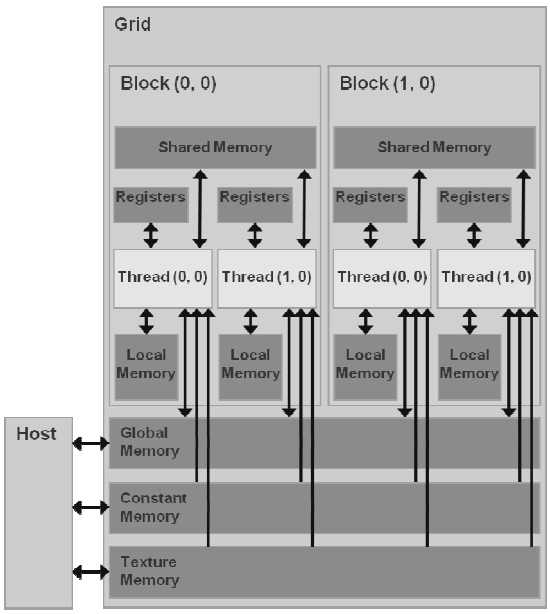

Figure 2: GPU Memory Model

- **Host Memory:** This is the main memory of the computer, from/to which data can be loaded/written back from the device memory.
- **Local Memory:** This memory is available to each thread running in the device.
- **Shared Memory:** This is shared between the various threads of a block. This is on-chip memory and hence the access is very fast. It is divided into various banks, which are equally sized memory modules.
- **Global Memory:** This memory is accessible to all the threads and blocks, and is usually used to load the host data into the GPU memory. As the memory is not cached, the access to this memory is not as fast, but a right access pattern can be used to maximize memory bandwidth.
- **Texture Memory:** This is a cached memory, hence is faster than global memory. The texture memory can be loaded by the host, but can be only read by the device kernel. It also provides normalized access to the data. Useful for reading images in the kernel.
- **Constant Memory:** This is also a cached memory for fast access to constant data.

2.3 Hardware Implementation

- GPU consists of an array of multiprocessors, such that threads of a thread block run concurrently on one multiprocessor. As the blocks finish, new blocks are launched on the vacated blocks. The overall device architecture can be shown as:

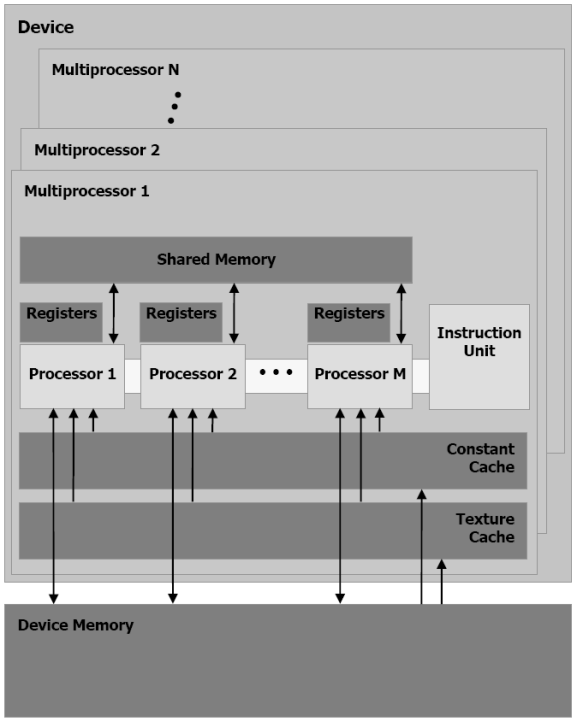

Figure 3: Multiprocessors on GPU

- Each multiprocessor executes the threads in *wraps*, which are groups of 32 parallel threads.
- Thus each multiprocessor consists of local *registers*, *shared memory* that is shared by all the scalar processors and a *constant* cache. *Texture* Cache is available through a *texture unit*.
- The size of block is limited by the amount of registers required per thread and the amount of shared memory. Kernel fails to launch if not even a single block can be launched on a multiprocessor.

2.4 Few important extensions in CUDA

- **Function Type Qualifiers:**
  - **`__global__`** declares a function as kernel. The function is executed on device and can be called from host.
  - **`__device__`** declares a function which executed on device and called from host.
  - **`__host__`** used to identify a function executed and called from host only.
- **Variable type qualifiers:**
  - **`__device__`** defines a variable stored in device memory. It resides in global memory space and accessible from all the threads.
  - **`__constant__`** declares a variable in constant memory space.
  - **`__shared__`** declares a variable in shared memory space of thread block.

- **Built-in variables**
  - **`gridDim`:** stores the dimensions of the grid, is a 3-component vector.
  - **`blockIdx`:** stores the block index within the grid as a 3-component vector.
  - **`blockDim`:** stores the dimensions of a block.
  - **`threadIdx`:** stores the thread index within the block as a 3-component vector.
- **Run-time APIs**
  - **`cudaMalloc`:** allocates memory, of the size given as input, in the global memory space of the device. Similar to **`malloc`** in C.
  - **`cudaFree`:** frees the memory allocated by **`cudaMalloc`**.
  - **`cudaMemcpy`:** copies data to/from device memory from/to host memory.

3. Bilateral Filters on GPU

3.1 Introduction

Bilateral filters were first coined by Tomasi and Manduchi [1]. These filters smoothen the image but keep the edges constant by means of non-linear combination of the image values of nearby pixels. This has been achieved by a combination of *range* filtering and *spatial* filtering. Range filters operate on value of the image pixels rather than their location; spatial filters take the location into account. By combining both of them, the paper achieves an edge-sensitive smoothening filter, which varies both according to the image pixel value as well location. A general form of the filter is given by:

$$h(x) = k(x)^{-1} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\mathcal{E}) c(\mathcal{E}, x) s\big(f(\mathcal{E}), f(x)\big) d\mathcal{E}$$

$$\text{Where, } k(x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\mathcal{E}, x) s\big(f(\mathcal{E}), f(x)\big) d\mathcal{E}$$

Here, c(.) is the geometric distance between the pixels x and Ɛ, and s(.) is a similarity function which measures how close the value of the image pixel is to the given value. For the special case of Gaussian c(.) and s(.), the equation becomes:

$$c(\mathcal{E}, x) = e^{-\frac{1}{2}\left(\frac{\left|\left|\mathcal{E}-x\right|\right|^2}{\sigma_d^2}\right)}$$

$$s\big(f(\mathcal{E}), f(x)\big) = e^{-\frac{1}{2}\left(\frac{\left|\left|f(\mathcal{E})-f(x)\right|\right|^2}{\sigma_r^2}\right)}$$

The functioning of the bilateral filter can be seen in the following figures:

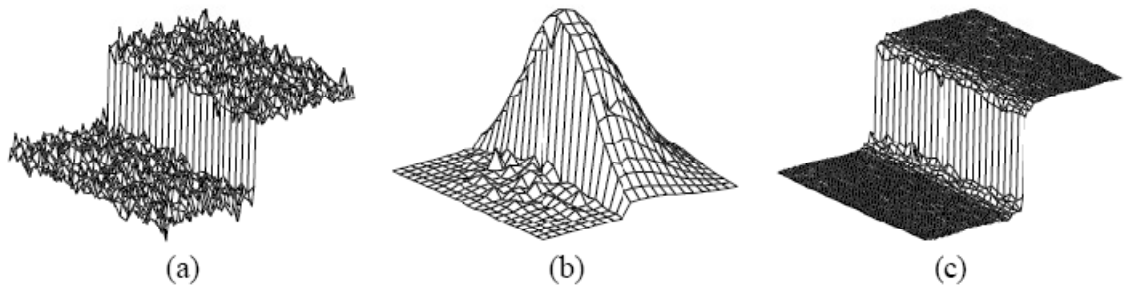

Figure 4: (a) A step function perturbed by random noise (b) combined similarity weights a pixel right to the step (c) final smoothened output [1]

3.2 Implementation

Bilateral filter cannot be implemented by using the FFT algorithms in this form, because the filter values change with image pixel location, depending on the image values of neighbouring pixels. Moreover it is also not separable in its current form. Brute –force algorithm was used to implement the filter on both GPU and CPU. The pseudo-code for the algorithm can be given as:

For input: image I, Gaussian Parameters σ_d and σ_r, output image I^b, W^b weight coefficients

1. All values of I^b and W^b initialized to zero.

2. For each pixel (x, y) with intensity I(x,y)
   a. For each pixel (x', y') in image with values I(x',y')
      Compute the associated weight:
      weight $\leftarrow \exp(-(I(x',y') - I(x,y))^2/2\sigma^2_d - ((x - x')^2 + (y - y')^2)/2\sigma^2_s)$
   b. Update the weight sum $W^b(x,y) = W^b(x,y) + \text{weight}$
   c. Update $I^b(x,y) = I^b(x,y) + \text{weight} \times I^b(x',y')$

3. Normalize the result:

   $I^b(x,y) \leftarrow I^b(x,y)/ W^b(x,y)$

For actual implementation, the filter radius was taken to be twice the value of its spatial sigma, as the Gaussian tail dies off quickly. This truncated filter was used as an approximation for the full kernel.

3.3 GPU Implementation

For GPU implementation, the following template was followed:

```
{
// load image from disk
// load reference image from image (output)
// allocate device memory for result
// allocate array and copy image data
// set texture parameters
 // access with normalized texture coordinates
// Bind the array to the texture

dim3 dimBlock(8, 8, 1);
dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);

// execute the kernel
BilateralFilter<<< dimGrid, dimBlock, 0 >>>( image, spat_filter, width, height, sigmar, sigmad);
 // check if kernel execution generated an error
 // allocate mem for the result on host side
 // copy result from device to host
 // write result to file
 // cleanup memory}
```

Some of the optimizations used in the code are:

- Texture memory has been used for accessing the image values. Texture memory being cache memory provides a fast access to the image data.
- Spatial filter was calculated in the host code, and passed to the kernel as a constant matrix. This reduced the time for computing the values again for every pixel.

NVIDIA 8600 graphics card was used to implement the codes.

3.4 CPU Implementation

CPU code was similar to the GPU code except that the bilateral filter function was executed using for …. then loop over all the pixels of the image, which run in parallel threads in GPU. This was done to get a better estimate of the CPU and GPU timings, as they are running same algorithm. The CPU under test was Intel Quad Core Processor running at 2.4 GHz.

3.5 Speed Comparison

A 512 x 512 gray scale Lena image was given as input to the program. The speed comparisons were made in two cases:

Varying σ_d keeping σ_r constant

Results for various sigma values are tabulated below for $\sigma_r = 0.1$

Spatial sigma (σ_d)	GPU Time (ms)	CPU Time (ms)	Speed GPU (Mpix/s)	Speed CPU (Mpix/s)	Ratio
1	230	1880	1.14	0.14	8.2
2	290	7310	0.90	0.036	25.2
3	330	16390	0.79	0.016	50
4	400	29010	0.66	0.009	72.5
5	520	45130	0.50	0.005	87
6	660	65200	0.40	0.004	99

Thus, we can see that CPU is much slower than GPU in executing the same task. Further, the time taken for CPU increases in approximately n^2 fashion with increase in the filter length. Hence, the ratio of speeds increases with increase in filter length and reaches at about 100x in the last case.

Varying σ_r keeping σ_d constant

The range sigma was also varied keeping spatial sigma constant. The time of execution for GPU and CPU was found to be almost constant for different values of σ_r.

Range sigma (σ_r)	GPU Time (ms) for $\sigma_d = 5$	CPU Time (ms) for $\sigma_d = 3$
0.1	518	16390
0.2	516	16360
0.3	512	16370
0.4	507	16320

Output Images:

- Comparison of CPU and GPU output images:

Original Image

GPU output for $\sigma_d = 3$ $\sigma_r = 1$

CPU output for $\sigma_d = 3$ $\sigma_r = 1$

Difference Image

- Variation with σ_r, keeping spatial sigma, σ_d constant (GPU outputs):

$\sigma_d = 3$ $\sigma_r = 0.1$

$\sigma_d = 3$ $\sigma_r = 0.3$

$\sigma_d = 5$ $\sigma_r = 0.6$

- Variation with σ_d, keeping range sigma σ_r constant (GPU outputs):

$\sigma_d = 1$ $\sigma_r = 0.1$

$\sigma_d = 5$ $\sigma_r = 0.3$

$\sigma_d = 10$ $\sigma_r = 0.6$

4. Conclusions

- GPU was able to achieve a much better time response than CPU in all the cases of filter implementation. The ratio of speeds increased with increase in filter length.
- The error between the GPU and CPU outputs was very loss, thus GPU performs the calculations quite accurately.
- Variation in spatial sigma and range sigma showed desirable changes in the output image. Increase in range sigma value, keeping the other constant increased the blurring across the edges as expected. Similarly, keeping the value of range sigma constant and increasing the other value resulted in better smoothening of the images without disturbing the edges.

5. Future Work

- The majority of the first stage work was exploratory, learning about the architecture of GPU and learning to implement CUDA language. I also gave a basic demonstration of bilateral filtering in GPU.
- Many fast approaches have been developed to implement bilateral filter. These can be implemented in GPU and the performance can be improved further.
- Further, more complex problems can be implemented, which would require further exploring the capabilities of GPU.
- A comparative study of different GPU platforms can also be made in testing the algorithms.

6. References

1. *C. Tomasi, R. Manduchi:* Bilateral Filtering for gray and colour images, *IEEE International Conference on Computer Vision, 1998*.
2. CUDA Programming Guide, NVIDIA