Advanced Computer Architecture

Final Project Report

GPU (CUDA) Image Filters

Preshit Harlikar
Smitesh Modak

August 12, 2018

Overview

The project aims at implementing Filters such as Sobel, Average and Boost Filter in CUDA on both CPU and GPU to determine the performance of the computation/algorithm. CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers can dramatically speed up computing applications by harnessing the power of GPUs.

The Sobel operator is used in image processing, particularly within edge detection algorithms. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is either the corresponding gradient vector or the norm of this vector. The Sobel operator is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical direction and is therefore relatively inexpensive in terms of computations. On the other hand, the gradient approximation which it produces is relatively crude, for high frequency variations in the image.

Mean filtering is a simple, intuitive and easy to implement method of smoothing images, i.e. reducing the amount of intensity variation between one pixel and the next. It is often used to reduce noise in images. The idea of mean filtering is simply to replace each pixel value in an image with the mean (`average') value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter. Like other convolutions it is based around a kernel, which represents the shape and size of the neighborhood to be sampled when calculating the mean. Often a 3×3 square kernel is used, as shown in Figure 1, although larger kernels (e.g. 5×5 squares) can be used for more severe smoothing.

The high-boost filter can be used to enhance high frequency component while keeping the low frequency components. High boost filter is composed by an all pass filter and a edge detection filter (Laplacian filter). Thus, it emphasizes edges and results in image sharpener. The high-boost filter is a simple sharpening operator in signal and image processing. It is used for amplifying high frequency components of signals and images. The amplification is achieved via a procedure which subtracts a smoothed version of the media data from the original one.

Our results show that a pure GPU optimization of the algorithm achieved a factor of 50-60 times improvement over the CPU version. These results show that GPUs are not replacements for CPU architecture. Rather, they are powerful accelerators for existing infrastructure. GPU-accelerated computing offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications just run much faster. While general-purpose computing is still the CPU's domain, GPUs are the hardware backbone of nearly all intensive computational applications.

Algorithm

Sobel Filter:

Mathematically, the gradient of a two-variable function (here the image intensity function) is at each image point a 2D vector with the components given by the derivatives in the horizontal and vertical directions. At each image point, the gradient vector points in the direction of largest possible intensity increase, and the length of the gradient vector corresponds to the rate of change in that direction. This implies that the result of the Sobel operator at an image point which is in a region of constant image intensity is a zero vector and at a point on an edge is a vector which points across the edge, from darker to brighter values.

Mathematically, the operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

where * here denotes the 2-dimensional convolution operation. The x-coordinate is here defined as increasing in the "right"-direction, and the y-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

Using this information, we can also calculate the gradient's direction:

$$\mathbf{\Theta} = \operatorname{arctan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

where, for example, Θ is 0 for a vertical edge which is darker on the left side.

Essentially the angle Θ will indicate whether there is a gradient (difference) and could be 360 degrees to indicate the weighted difference of the 8 surrounding pixels.

At the same time, there is an approximation that says if if either Gx or Gy have substantial value, then the pixel point is located at an edge of the image. Otherwise, if a pixel is surrounded by all identical pixel values (8 neighbors), then Gx and Gy will be zero. You should double check that you understand this by seeing that the values of each 3x3 matrices are designed to cancel each other out (negatives on a column would cancel out the positive weights on the other, same for the matrix involving row values).

The Sobel operator represents a rather an approximation of the image gradient but is still of sufficient quality to be of practical use in many applications. More precisely, it uses intensity values only in a 3×3 region around each image point to approximate the corresponding image gradient, and it uses only integer values for the coefficients which weight the image intensities to produce the gradient approximation.

Pseudo Code:

- *Input*: A Sample Image.
- *Output*: Detected Edges.
- *Step 1*: Accept the input image.
- *Step 2*: Apply mask Gx, Gy to the input image.
- *Step 3*: Apply Sobel edge detection algorithm and the gradient.
- *Step 4*: Masks manipulation of Gx, Gy separately on the input image.
- *Step 5*: Results combined to find the absolute magnitude of the gradient.
- *Step 6*: The absolute magnitude is the output edges.

Average Filter:

The mean filter is a simple sliding-window spatial filter that replaces the center value in the window with the average (mean) of all the pixel values in the window. The window, or kernel, is usually square but can be any shape. An example of mean filtering of a single 3x3 window of values is shown below.

unfiltered values

5	3	6
2	1	9
8	4	7

5 + 3 + 6 + 2 + 1 + 9 + 8 + 4 + 7 = 45
45 / 9 = 5
.

mean filtered

*	*	*
*	5	*
*	*	*

Center value (previously 1) is replaced by the mean of all nine values (5).

High Boost Filter:

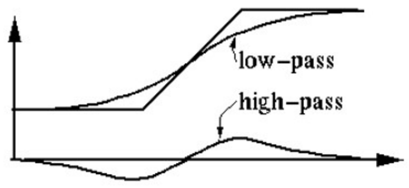
The high-boost filter can be used to enhance high frequency component while keeping the low frequency components:

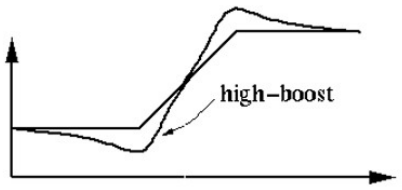
$$I_{hb} = I_o + c\, I_{hp} = (W_{ap} + c\, W_{hp}) * I_o = W_{hb} * I_o$$

where c is a constant and $W_{hb} = cW_{ap} + W_{hp}$ is the high boost convolution kernel. For example:

$$W_{hb} = W_{ap} + c\,W_{hp} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + c \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -c & 0 \\ -c & 4c+1 & -c \\ 0 & -c & 0 \end{bmatrix}$$

$$W_{hb} = W_{ap} + c\,W_{hp} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + c \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -c & -c & -c \\ -c & 8c+1 & -c \\ -c & -c & -c \end{bmatrix}$$





The example below shows the effect of high-boost filtering obtained by the above high-boost convolution kernel with $c = 2$. The image on the left is the original image, the one in the middle is high-boost filtered. Note that the low spatial frequency components (global, large black background and bight areas) are suppressed while the high spatial frequency components (the texture of the fur and the whiskers) are enhanced. After a linear stretch, the image on the right is obtained.

Implementation

CPU Implementation for Sobel Filter:

```
void CPU_Sobel(unsigned char* imageIn, unsigned char* imageOut, int width, int
height)
{
	int i, j, rows, cols, startCol, endCol, startRow, endRow;
	const float SobelMatrix[9] = {-1,0,1,-2,0,2,-1,0,1};

	rows = height;
	cols = width;

	// Initialize all output pixels to zero
	for(i=0; i<rows; i++)
	{
		for(j=0; j<cols; j++)
		{
			imageOut[i*width + j] = 0;
		}
	}

	startCol = 1;
	endCol = cols - 1;
	startRow = 1;
	endRow = rows - 1;

	// Go through all inner pizel positions
	for(i=startRow; i<endRow; i++)
	{
		for(j=startCol; j<endCol; j++)
		{
			// sum up the 9 values to calculate both the direction x and
			direction y
			float sumX = 0, sumY=0;
			for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++) {
			for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++) {
			float Pixel = (float)(imageIn[i*width + j +  (dy * width +
			dx)]);
			sumX += Pixel * SobelMatrix[(dy + FILTER_RADIUS) *
			FILTER_DIAMETER + (dx+FILTER_RADIUS)];
			sumY += Pixel * SobelMatrix[(dx + FILTER_RADIUS) *
			FILTER_DIAMETER + (dy+FILTER_RADIUS)];
			}
			}
			imageOut[i*width + j] = (abs(sumX) + abs(sumY)) >
			EDGE_VALUE_THRESHOLD ? 255 : 0;
		}
	}
}
```

GPU Kernel Implementation for Sobel Filter:

```
__global__ void SobelFilter(unsigned char* g_DataIn, unsigned char* g_DataOut,
int width, int height)
{
   __shared__ unsigned char sharedMem[BLOCK_HEIGHT * BLOCK_WIDTH];
   float s_SobelMatrix[9]= {-1,0,1,-2,0,2,-1,0,1};

   // Computer the X and Y global coordinates
   int x = blockIdx.x * TILE_WIDTH + threadIdx.x ;//- FILTER_RADIUS;
   int y = blockIdx.y * TILE_HEIGHT + threadIdx.y ;//- FILTER_RADIUS;

   // Get the Global index into the original image
   int index = y * (width) + x;

   // STUDENT:  Check 1
   // Handle the extra thread case where the image width or height
   //
   if (x >= width || y >= height)
      return;

   // STUDENT: Check 2
   // Handle the border cases of the global image
   if(x < FILTER_RADIUS || y < FILTER_RADIUS)
   {
       g_DataOut[index] = g_DataIn[index];
       return;
   }

   if ((x > width - FILTER_RADIUS - 1)&&(x <width))
   {
       g_DataOut[index] = g_DataIn[index];
       return;
   }

   if ((y > height - FILTER_RADIUS - 1)&&(y < height))
   {
       g_DataOut[index] = g_DataIn[index];
       return;
   }

   // Perform the first load of values into shared memory
   int sharedIndex = threadIdx.y * blockDim.y + threadIdx.x;
   sharedMem[sharedIndex] = g_DataIn[index];
   __syncthreads();
```

```
    // STUDENT: Make sure only the thread ids should write the sum of the
       neighbors.
    float sumX = 0, sumY=0;
    for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++)
    {
        for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++)
        {
            float PIXEL = (float) (g_DataIn[y*width +x + (dy*width + dx)]);
            sumX += PIXEL * s_SobelMatrix[(dy + FILTER_RADIUS) *
            FILTER_DIAMETER + (dx + FILTER_RADIUS)];
            sumY += PIXEL * s_SobelMatrix[(dx + FILTER_RADIUS) *
            FILTER_DIAMETER + (dy + FILTER_RADIUS)];
        }
    }

    g_DataOut[index] =(abs(sumX) + abs(sumY)) > EDGE_VALUE_THRESHOLD ? 255 : 0;
}
```

GPU Kernel Implementation for Average Filter:

```
__global__ void AverageFilter(unsigned char* g_DataIn, unsigned char*
g_DataOut, int width, int height)
{
    __shared__ unsigned char sharedMem[BLOCK_HEIGHT * BLOCK_WIDTH];

    float AverageMatrix[9]={1,1,1,1,1,1,1,1,1};

    int x = blockIdx.x * TILE_WIDTH + threadIdx.x ;//- FILTER_RADIUS;
    int y = blockIdx.y * TILE_HEIGHT + threadIdx.y ;//- FILTER_RADIUS;

    // Get the Global index into the original image
    int index = y * (width) + x;

    // STUDENT:  Check 1
    // Handle the extra thread case where the image width or height
    //
    if (x >= width || y >= height)
       return;

    // STUDENT: Check 2
    // Handle the border cases of the global image
    if( x < FILTER_RADIUS || y < FILTER_RADIUS)
    {
        g_DataOut[index] = g_DataIn[index];
        return;
    }

    if ((x > width - FILTER_RADIUS - 1)&&(x <width))
    {
        g_DataOut[index] = g_DataIn[index];
        return;
    }
```

```
   if ((y > height - FILTER_RADIUS - 1)&&(y < height))
   {
       g_DataOut[index] = g_DataIn[index];
       return;
   }

   // Perform the first load of values into shared memory
   int sharedIndex = threadIdx.y * blockDim.y + threadIdx.x;
   sharedMem[sharedIndex] = g_DataIn[index];
   __syncthreads();


   // STUDENT: Make sure only the thread ids should write the sum of the
      neighbors.
   float sumX = 0;
   for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++)
   {
       for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++)
       {
           float PIXEL = (float) (g_DataIn[y*width + x + (dy*width + dx)]);
           sumX += PIXEL * AverageMatrix[(dy + FILTER_RADIUS) *
           FILTER_DIAMETER + (dx + FILTER_RADIUS)];
       }
   }

   g_DataOut[index] = sumX/9;
}
```

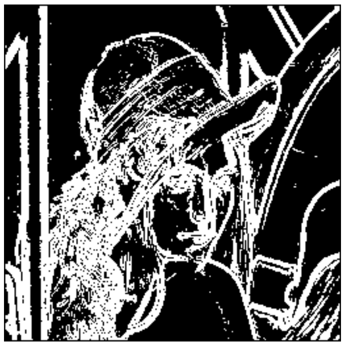
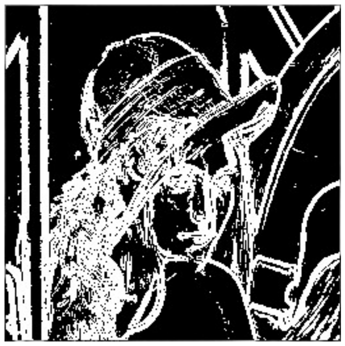
GPU Kernel Implementation of High Boost Filter:

```
__global__ void HighBoostFilter(unsigned char* g_DataIn, unsigned char*
g_DataOut, int width, int height)
{
   __shared__ unsigned char sharedMem[BLOCK_HEIGHT * BLOCK_WIDTH];

   float HighBoostMatrix[9]={1,1,1,1,1,1,1,1,1};
   float PIXEL;

   int x = blockIdx.x * TILE_WIDTH + threadIdx.x ;//- FILTER_RADIUS;
   int y = blockIdx.y * TILE_HEIGHT + threadIdx.y ;//- FILTER_RADIUS;

   // Get the Global index into the original image
   int index = y * (width) + x;

   // STUDENT:  Check 1
   // Handle the extra thread case where the image width or height
   //
   if (x >= width || y >= height)
      return;
```

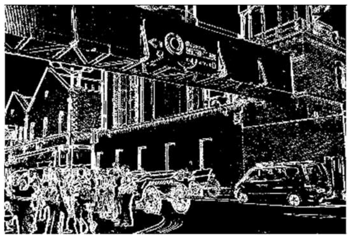
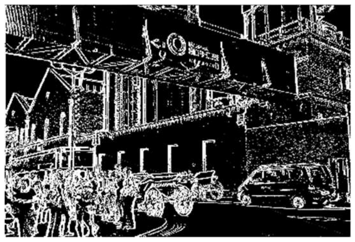
```
    // STUDENT: Check 2
    // Handle the border cases of the global image
    if( x < FILTER_RADIUS || y < FILTER_RADIUS)
    {
        g_DataOut[index] = g_DataIn[index];
        return;
    }

    if ((x > width - FILTER_RADIUS - 1)&&(x <width))
    {
        g_DataOut[index] = g_DataIn[index];
        return;
    }

    if ((y > height - FILTER_RADIUS - 1)&&(y < height))
    {
        g_DataOut[index] = g_DataIn[index];
        return;
    }

    // Perform the first load of values into shared memory
    int sharedIndex = threadIdx.y * blockDim.y + threadIdx.x;
    sharedMem[sharedIndex] = g_DataIn[index];
    __syncthreads();


    // STUDENT: Make sure only the thread ids should write the sum of the
       neighbors.
    float sumX = 0;
    for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; dy++)
    {
        for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; dx++)
        {
            PIXEL = (float) (g_DataIn[y*width + x + (dy*width + dx)]);
            sumX += PIXEL * HighBoostMatrix[(dy + FILTER_RADIUS) *
            FILTER_DIAMETER + (dx + FILTER_RADIUS)];
        }
    }

    g_DataOut[index] = CLAMP_8bit((int)(PIXEL + HIGH_BOOST_FACTOR *
    (uint8_t)(PIXEL - sumX/9)));
}
```

Results

Sobel Filter Output:

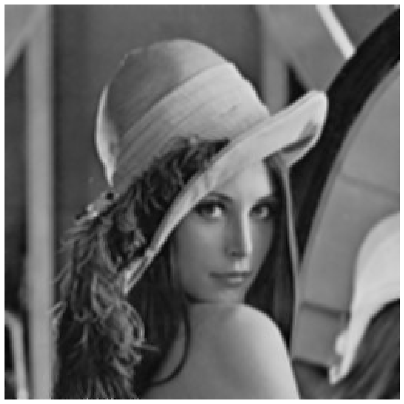



Original Images CPU Sobel Images GPU Sobel Images

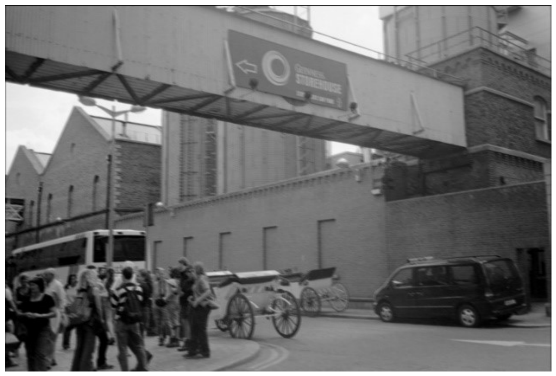
Input Size	GPU Execution Time (us)	CPU Execution Time (us)	Speedup (us)
lena.bmp	607	35706	35099
dublin.bmp	32051	296722	264671

__Average Filter Output:__

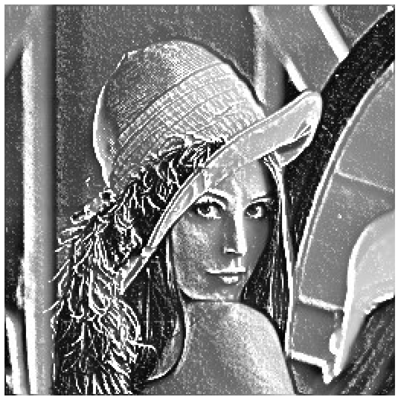


Original Images

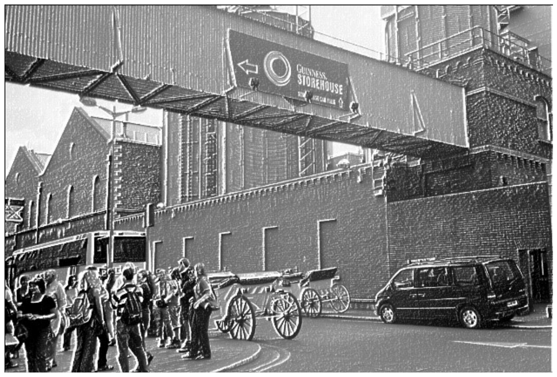
Average Filtered Images



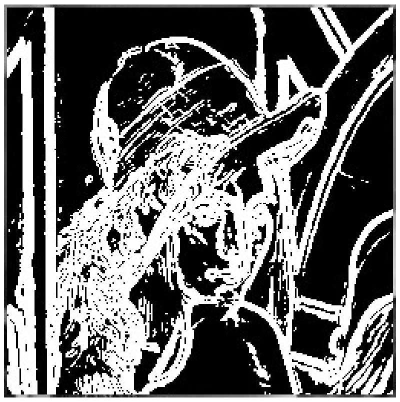
__High Boost Filter Output:__



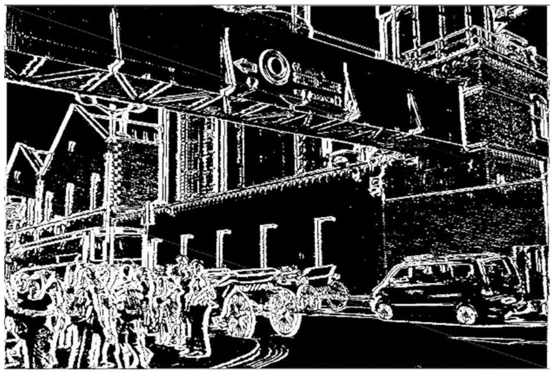
Original Images High Boost Filtered Images



Sobel Filter 5x5 Output:



Original Images Sobel 5x5 Filter Output Images



Input Size	3x3 GPU Execution Time (us)	5x5 GPU Execution Time (us)	Speedup (us)
lena.bmp	607	493	114
dublin.bmp	32051	18562	13489

References:

[1]https://pdfs.semanticscholar.org/6bca/fdf33445585966ee6fb3371dd1ce15241a62.pdf
[2]https://developer.nvidia.com/cuda-zone
[3]https://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.html
[4]https://www.scribd.com/doc/119044470/High-Boost-Filtering
[5]https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4
[6]https://pdfs.semanticscholar.org/6bca/fdf33445585966ee6fb3371dd1ce15241a62.pdf
[7]https://www.markschulze.net/java/meanmed.html
[8]https://fourier.eng.hmc.edu/e161/lectures/gradient/node2.html