ELEC 4723/5723 High Performance Computer Architecture (HPCA)

HW: GPU (CUDA) Image Filters

Topics Covered:

* Shared memory utilization: allocation and shared access
* 2-D task description of blocks
* Reformulation of computation for data-level parallelism

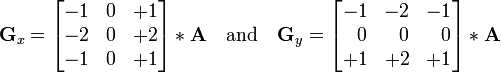
**Overview:**

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, in particular for high frequency variations in the image.

In simple terms, the operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result therefore shows how "abruptly" or "smoothly" the image changes at that point, and therefore how likely it is that that part of the image represents an edge, as well as how that edge is likely to be oriented. In practice, the magnitude (likelihood of an edge) calculation is more reliable and easier to interpret than the direction calculation.

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](http://en.wikipedia.org/wiki/Convolution) 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:



where \* here denotes the 2-dimensional [convolution](http://en.wikipedia.org/wiki/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:



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



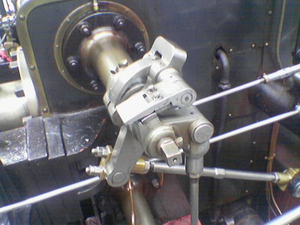
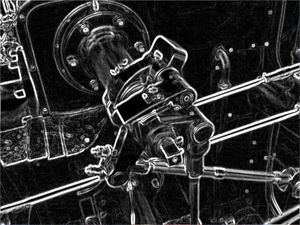
where, for example, Θ is 0 for a vertical edge which is darker on the left side.

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.

Sample sobel operator on the test image:



CPU Code for Sobel Filter (uses the weight of Gx and Gy to determine setting output point as gradient ON (255), off (0).

|  |
| --- |
| #define RADIUS 1  #define DIAMETER 3  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 x and y direction  float sumX = 0, sumY=0;  for(int dy = -FILTER\_RADIUS; dy <= RADIUS; dy++) {  for(int dx = -FILTER\_RADIUS; dx <= RADIUS; dx++) {  float Pixel = (float)(imageIn[i\*width + j + (dy \* width + dx)]);  sumX += Pixel \* SobelMatrix[(dy + RADIUS) \* DIAMETER + (dx+RADIUS)];  sumY += Pixel \* SobelMatrix[(dx + RADIUS) \* DIAMETER + (dy+RADIUS)];  }  }  imageOut[i\*width + j] = (abs(sumX) + abs(sumY)) > THRESHOLD ? 255 : 0;  }  }  } |

Each group is responsible accessing the GPU-enabled machine: allison1.cudenver.edu

[Part A] Use the code skeleton to write the code for the sobel filter. Parts of the provided code are provided for you as the zipped file: filter.zip

/home/usergroup1/NVIDIA\_CUDA\_SDK/C/src/

>> cd filter

change the code

>> make

remember that the binary program will be located at: ../../bin/linux/release/filter

You can run the program as the following:

filter –file lena.bmp –out sobel.bmp –filter sobel

By default, the CPU will also generate CPU\_sobel.bmp

You will also be designing the 3x3 filter: average and boost

filter –file lena.bmp –out average.bmp –filter average

filter –file lena.bmp –out boost.bmp –filter boost

[Part B] Use the timing routines to calculate the amount of time for CPU execution compared to GPU execution (include memory transfer time) for the Sobel filter.

[Handin – Upload to Blackboard]

You must upload the following to blackboard. Your results will be given points based on clarity and following what is asked.

[Item 1] - For the assignment, submit the .cu file and the

[Item 2] - Provide the execution time for the Sobel filter:

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | GPU Execution Time | CPU Execution Time | Speedup |
| lena.bmp |  |  |  |
| dublin.bmp |  |  |  |

[Item 3] – Implement the Average Filter

Similar to the Sobel filter, except each pixel generates the sum of eight neighbors and the center pixel and divides by 9. In the case of pixels as characters, it is best to cast the values as float, add a sum, and then cast the resulting float back to an unsigned character:

|  |  |  |
| --- | --- | --- |
| 1 | 3 | 5 |
| 2 | 7 | 3 |
| 6 | 4 | 1 |

1 + 3 + 5 + 2 + 7 + 3 + 6 + 4 + 1 = 32

average = 32 / 9 = 3

g\_DataOut[index] = (unsigned char)(sum / FILTER\_AREA);

[Item 4] – Implement the Boost Filter

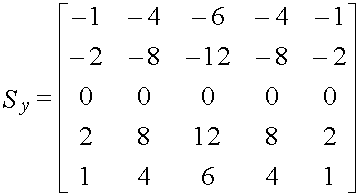
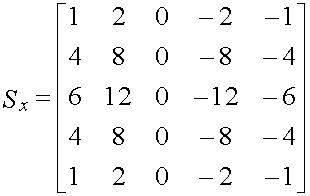
Similar to the average filter, the Boost filter sums the 9 values, and hen subtracts the difference between the center pixel and the average). The difference is then multiplied by a boost factor and added to the original value, finally the result is guaranteed to be between 0 and 255. In the case example above, the center pixel is 7, and with a boost factor of 10:

7-3 = 4, and the new pixel value would be 4+40 = 44.

g\_DataOut[index] = CLAMP\_8bit(centerPixel + HIGH\_BOOST\_FACTOR \* (unsigned char)(centerPixel - sum / FILTER\_AREA));

#define CLAMP\_8bit(x) max(0, min(255, (x)))

**[Item 4] – Graduate Students-** You must rewrite the sobel filter to accept a 5x5 array kernel, using the following fomats:



Your solution MUST pass the 5x5 array as an argument to the CUDA kernel. Currently the kernel uses a local variable array to implement the 3x3 filter kernel, which is very costly in terms of execution performance since each thread has an identical duplicated 3x3 array. Hint: FILTER\_RADIUS is now 3 and FILTER\_DIAMETER is 5.

You must submit your .cu to include a sobel5by5 kernel, and calculate the execution times and speedups:

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | 3x3 GPU Execution Time | 5x5 GPU Execution Time | Speedup |
| lena.bmp |  |  |  |
| dublin.bmp |  |  |  |