Parallelized Prewitt Algorithm Using CUDA

Aaron Daniel[#1], Kevin Hicks[#2], Ryan Marschall[#3]

College of Engineering, University of South Florida
4202 E Fowler Ave, Tampa, FL 33620

aarondaniel@mail.usf.edu

kevinhicks@mail.usf.edu

rmarschall@mail.usf.edu

***Abstract*— In this paper we will go over the design, implementation and performance of a parallelized Prewitt edge detection algorithm. This paper is made up of three sections, the first being a brief description of the Prewitt algorithm and its uses. Second being our implementation of the parallelized algorithm using Nvidia CUDA. And finally a discussion about performance of the GPU v. CPU filter and what conditions are ideal for both**

***Keywords*— Prewitt, CUDA, DIP, image processing, edge detection**

I. Introduction

As explained in the abstract this paper is an analysis of a parallelized version of the Prewitt edge detection algorithm. We understand that there are other parallelized implementations of edge detection algorithms, but none are implemented in one single .cu file. openCV is a famous image processing API that has CUDA functionality built in, but requires a ridiculous amount of setup time. So, our goal was pretty simple. Implement a Prewitt edged detection algorithm in a lightweight package. Let us be clear, we will not be testing our algorithm against the openCV implementation because a Prewitt operator is not a standard function. In order to make our final performance test fair we had to use a standard image importing library called *mypgm.h*[1]. This gave us predefined functions to both load and save .pgm files. PGM stands for Portable GrayMap format and is the go to format for image processing applications. The images pixels are represented as a massive grid of numbers. It is fairly obvious why this format is ideal for situations like this. From the same source we were able to adapt a similar edge detection algorithm into a functioning Prewitt CPU algorithm and we will use it as a baseline for our CUDA implementation.

II. Prewitt edge detection algorithm

The Prewitt edge detection algorithm is a gradient approximator using the image intensity function. The edges in the image are found using the delta between neighboring intensities. The intensities are determined and calculated using a mask which is a grid of numbers in either a horizontal or vertical orientation. The mask is a grid usually 3 by 3 and has a few different attributes. Weights on the outside of the mask must sum to zero, and weights must oppose each other

-1	0	1
-1	0	1
-1	0	1

**Vertical mask*

Note that the mask has a center column or row(horizontal mask) that is populated with zeros. This allows the window/mask to place itself over an edge and calculate the differences in pixel intensities on either side of the edge.

This gives us a very well defined edge and can be done in a recursive or looping manner.

The basic steps of this algorithm are to first convert a .PGM grayscale image into a 2d array. We use a grayscale image because in image processing the overall goals are computer vision, pattern recognition and general structure. Since an image's color values are represented in values ranging from 0 to 255 we really only want to deal with the limits. The color white has a pixel intensity value of 255 while black has a value of 0. The next step is to cycle through the rows and columns of the input image array and apply matrix operations between our mask and the current location we are at in the loop. The results of each operation are stored in a value known as the gradient value. The gradient values are only used in this first loop to aid in finding the max and min values. A max and min are determined and after every iteration of the for loop the results must go through a check statement to determine which int color value should be placed as max or min. Once we have max and min values we run our for loop again (setting the gradient value back to zero) with a similar function adjusting the result of the final gradient value with a value of 255(MAX_BRIGHTNESS) which then creates a new resulting image array.

The uses for this algorithm are limitless especially with the burgeoning fields of machine learning and automation. In a new world where self driving cars and advanced lifesaving medical imaging are highly sought after, we can clearly see how much of an impact on humanity this can and will be if improved.

III. Parallelized CUDA algorithm

We started out by researching the problem and we needed a way to pull the image files into our algorithm, so we used a fairly standard .h file called *mypgm.h*. This is boilerplate file that can accept .pgm files and convert them into 2d arrays. While this method works perfectly for numerous image processing filters, it had to be slightly modified for our uses. When attempting to create a CUDA kernel function we noticed that it is practically impossible to pass a 2d array from host to device and vice versa. We attempted to use something called cudaMallocPitch() and cudaMemcpy2D() which only served to confuse us. Finally we found a simple solution that seems very simple now. Since a 2d array is just a construct for ease of usability we decided to use a 1d array and call the row and columns as a product(array[row * col]). We made an int value inside main called sizeE that effectively allocates memory for a 2d array using the images x and y dimensions when using malloc/cudaMalloc. After we passed the array into the kernel things got a little tricky. The first order of business is allocating shared memory on the device dynamically. The reason for choosing dynamically instead of statically is that at compile-time, the size of the allocation needed is unknown. Since image sizes can vary, the amount of threads per block being used is changing dependent on the size of the image. After dynamically allocating the array, the shared memory for each block is initialized. Then, the edge detection process is started. By applying a 3x3 filter to the pixel values around the target pixel. After applying the filter, each pixel value is aggregated to the gradient value. The actual gradient value is then calculated by the equation 255 * (gradient - minimum) / (maximum - minimum) where the maximum and minimum are their respective pixel values encountered for the image. This value is then applied to the pixel being worked on.

IV. Results

The results pretty much speak for themselves when it comes to the performance numbers. It should be known that the CPU performance statistics are based off workstations in the C4 lab

at the University of South Florida. The CPU used in these test is a Intel Xeon E5-2603 @ 1.60GHz. The GPU is an NVIDIA GeForce GTX 960. The test inputs we used to check our algorithm were images with varying sizes and pixel densities. Below we display the original image on the left and the result on the right. The sizes of the images increase as we go down the page and we can easily see the defined edges in the lower resolution images(digit & baboon.pgm). The higher resolution photos(beach & high.pgm) are a little harder to see in this reports thumbnails. By zooming in on the photos we can clearly see the result and we also get a better sense of the extreme magnitude of these photos.

Fig. 1 digit.pgm(120X80)

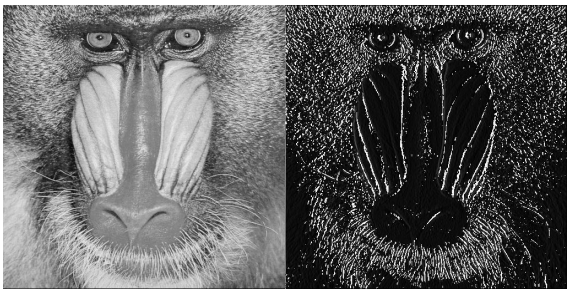

Fig. 2 baboon.pgm(512X512)

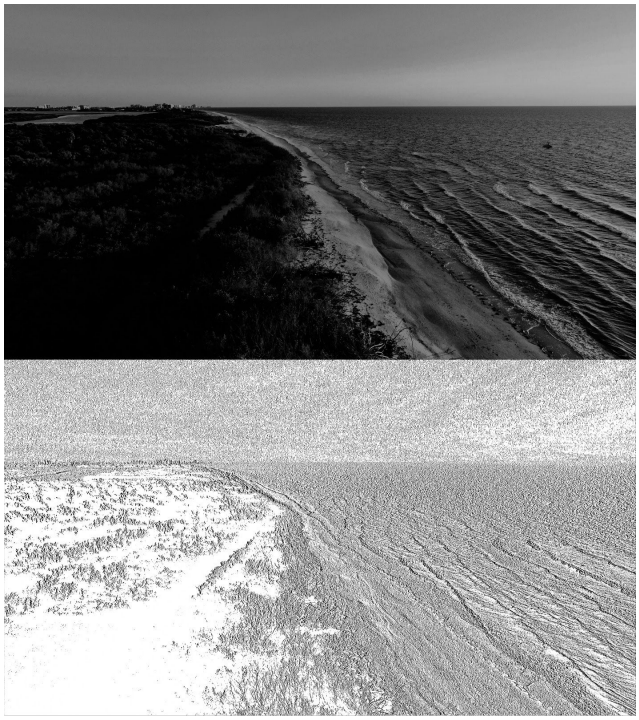

Fig. 3 beach.pgm(2250X4000)

Fig. 4 high.pgm(8494X8720)

As we can see the smaller the image the less of an impact parallelism has on the performance. This is generally expected because as the amount of threads required decreases, we have less of a need for split labor. The clock speed of a CPU core is usually substantially greater than that of a GPU core, yet there are far less of them to work with. The CPU core speeds can make up for the absence of cores, therefore

we get less of a performance gain at lower resolutions.

TABLE II
CPU v. GPU

	delay(10)	**digit.pgm**	**baboon.pgm**	**beach.pgm**	**high.pgm**
CPU	10000244	2736	45556	1426491	11716344
GPU		530	11750	380430	3051200

**All values in microseconds and CPU algorithm data found using[2]*

[1] T. Wakahara, pgmfile IO headerfile ,Computer and Information Sciences, Hosei University, Tokyo, Japan http://cis.k.hosei.ac.jp/~wakahara/mypgm.h

[2] T. Wakahara,Sobel.c ,Computer and Information Sciences, Hosei University, Tokyo, Japan http://cis.k.hosei.ac.jp/~wakahara/sobel.c