Parallelization of the Local Threshold and Boolean Function Based Edge Detection Algorithm Using CUDA

Raka JOVANOVIC
University of Belgrade
Institute of Physycs
Pregrevica 118, 11000 Belgrade
SERBIA
rakabog@yahoo.com

Milan TUBA
Megatrend University Belgrade
Faculty of Computer Science
Bulevar umetnosti 29, 11070 Belgrade
SERBIA
tuba@ieee.org*

Dana SIMIAN
Lucian Blaga University of Sibiu
Department of Computer Science
5-7 dr. I. Ratiu str., Sibiu
ROMANIA
dana.simian@ulbsibiu.ro

Abstract: In this paper we present a parallelized algorithm for edge detection for gray scale images. The chosen method is the local threshold and boolean function based edge detection. This method differs from common edge detectors in the use of bit map patterns instead of analyzing gradient changes in the image for edge recognition. The parallelization is implemented on the GPU, exploiting its multithreaded, many-core processor power using NVIDIA's CUDA (Compute Unified Device Architecture). We show in our tests the significant speedup of parallelized algorithm compared to the sequential one.

Key–Words: Image processing, Edge detection, Parallel algorithms, CUDA technology

1 Introduction

Edge detection is an important subject in image processing, due to its importance in a wide range of applications. For instance, edge detection is often used as a tool in pattern recognition algorithms. The problem of edge detection/recognition is an essential part of predictive image compression algorithms. When using one of these methods the prediction error is very high on edges if they are not recognized which greatly reduces the archived compression ratio.

Because of its importance several approaches have been developed for solving this problem. Edge detection can be developed for gray scaled or colored images. In this article we will focus on algorithms for gray scaled images. The Marr-Hildreth Edge Detector is a gradient based operator which uses the Laplacian to take the second derivative of an image [1]. Edge detection methods have also been developed using different gradient operators like Sobol, Roberts Cross and Prewitt [2]. The Canny Edge Detector [3] views edge detection as a signal processing optimization problem for which an objective function is defined. It is considered as a de facto standard for edge detection. Both of these approaches in a way analyze the change on the image like a gradient and consider large changes as edges. A completely different concept is used in the local threshold and boolean function based edge detection [4]. In that case 3x3 blocks of the image are converted in bit blocks. Some patterns appearing in the bit blocks are considered as indicators of edge existence. Additional research on this topic is in [5], [6].

A common property of all these algorithms is that they process a significant amount of data, all the pixels in the image. The necessary calculations for individual pixels can be done independently to a large extent. Because of these properties they are ideal for parallel implementation. Massive parallelization of algorithms has become very popular in the recent years with the development of the programmable Graphic Processor Unit or GPU, that has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational power and very high memory bandwidth. Several tools have been developed for developing software exploiting the power of the GPU like NVIDIA's CUDA (Compute Unified Device Architecture) [7], Khronos Group's OpenCL (Open Computing Language) [8] and Microsoft DirectCompute which is a part of Microsoft DirectX [9].

CUDA has been previously used in implementing a Sobol gradient based edge detector, Canny edge detector [10] and Gabor wavelet transform edge detection [11]. In this article we give details of a CUDA based implementation of the local threshold and boolean function based edge detection. We show the impressive speed improvement compared to the sequential algorithm.

The paper is organized as follows. In the second section we present the local threshold and boolean

*This research is supported by Ministry of Science, Republic of Serbia, Project No. III-44006

function based edge detection algorithm. In the third section we give details of its parallelization and implementation using CUDA. In the last section tests and results are presented.

2 The Local Threshold and Boolean Function Based Edge Detection

This edge detector has a greatly different approach than the most commonly used Canny's edge detection method. Canny's method and a wide range of similar ones relay on the analysis of gradient properties to determine the existence of edges, usually in combination with Gaussian smoothing for removing noise in the image.

The Local Threshold and Boolean Function Based Edge Detector does not rely on gradient changes, but it converts a window of pixels into a binary pattern based on a local threshold, and then applies masks to determine if an edge exists at a certain point or not. More precisely a pixel is considered an edge if the acquired binary pattern corresponds to an edge shape. Because the threshold is calculated on a per pixel basis, the edge detector should be less sensitive to variations in lighting throughout the picture. An advantage of this method is that it does not need any kind of smoothing to reduce noise in the image. It instead looks at the variance on a local level. The algorithm can be divided in the following 4 stages:

Stage 1. Convert a 3x3 window corresponding to a pixel of the image into a bit pattern using a local threshold. The local threshold value is recalculated for every pixel in the image as the mean of the 9 intensity values of the pixels in the window minus some small tolerance value. Conversion of the 3x3 window is done in the following way: if a pixel has an intensity value greater than this threshold, it is set to a 1 otherwise to zero. This gives a 3x3 binary pattern that we further analyze.

Stage 2. For each pixel, we compare its 3x3 binary pattern to patterns that correspond to edges. There are sixteen possible edge-like patterns that can exist for a 3x3 binary matrix, the patterns can be seen in Figure 1. If the binary window obtained in stage 1 is equivalent to any of these sixteen matrices, the center pixel of the window is considered to be an edge pixel.

Stage 3. Repeat stages 1 and 2 for every pixel in the image as the center pixel of the window. This way all the edges are acquired. The problem at this stage is that it will also give some false edges as a result of noise.

Stage 4. The final step is using a global threshold to remove false edges. The calculated variance for each 3x3 window, should have a maximum at an edge. This value is compared with a global threshold based on some image properties like the noise intensity. If this value of some pixel, expected to be an edge, is greater than the threshold, it is kept as an edge, otherwise it is removed.

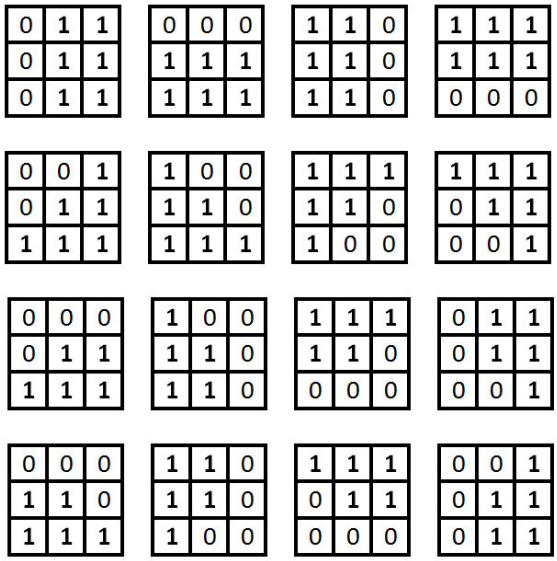

Figure 1: Patterns of 3x3 blocks that indicate that the center pixels is an edge

An interesting fact is that the boolean edge detector performs surprisingly similarly to the Canny edge detector even though the two methods use drastically different approaches. Canny's method is still preferred since it produces single pixel thick, continuous edges. The Boolean edge detector's edges are often spotty. In the analysis of the boolean edge detector performance it has been show that it performs better than Canny's algorithm on computer generated images that have sharp edges.

3 Implementation using CUDA

It is evident that, in the algorithm presented in the previous section, the calculation performed on individual pixels, are highly independent. Because of this fact it is very suitable for parallelization. CUDA C extends C by making it possible for a programmer to define kernel functions that, when called, are executed in parallel by different CUDA threads, contrary to only once in regular C functions. Because of this CUDA is ideal for implementing this type of algorithms. The architecture of the GPU is different compared to the CPU in a way that each of the processing nodes has a much smaller amount of available high speed memory and due to the fact that they have been constructed for

ISBN: 978-1-61804-089-3

arithmetic processing of the data. This exactly corresponds to the needs of an edge detection algorithm. In this section we give a detailed explanation how to implement this type of algorithm using CUDA.

It is obvious that the best way of parallelization is to create separate threads for every pixel in the image, perform the necessary calculations and store the information of edge existences. We can divide the implementation into two parts, the host function and the kernel functions. The host function is charged with starting threads and making the input/output data available to the device (GPU) since it has separate memory from the host (CPU). The kernel function is responsible for calculations, memory management on the device and thread synchronization. The host function implements the following steps:

1. Load InputImage to Host memory
2. Allocate InputImage memory on device
3. Copy InputImage from Host to device Memory
4. Start kernels
5. Copy OutputImage from device memory to host
6. Free allocated host and device memory.

In CUDA a large number of separate threads can be executed in parallel. Groups of threads are organized in blocks, and all the blocks are organized in a grids. This is done to make calculations more efficient in the sense of efficient use of available memory. In CUDA there is local memory dedicated to individual threads, shared memory that is used by threads in a block and global memory is available to all the threads. Shared memory is expected to be much faster than the global one; because of this kernel function can significantly improve performance if a majority the data needed for threads can be accessed from shared and local memory.

For convenience, all threads in CUDA have *ThreadIdx*, a three dimensional vector, that makes it possible for individual threads to identify themselves in a one, two or three dimensional blocks. Similarly a *BlockIdx* is used to identify a block inside of a grid. This makes implementation of an image processing filter like an edge detector easy, since we can connect a thread with a pixel*(X,Y)* in the image in the following way:

$$blockIdx.x * blockDim.x + threadIdx.x = X \quad (1)$$
$$blockIdx.y * blockDim.y + threadIdx.y = Y \quad (2)$$

In the local threshold and boolean function based edge detector, each thread needs to have access to the value of the corresponding pixel in the image and all of its neighbors. As previously mentioned, shared memory should be faster than the global one; because of this our kernel function needs to have the required pixel values available from the local memory. In Figure 2 we can see an example of pixels being processed and the values needed to be stored in the shared memory.

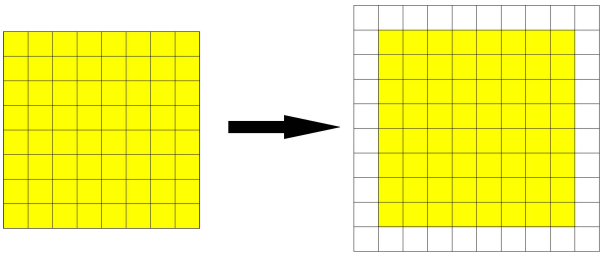

Figure 2: Left: image section that is being processed by thread block, Right: Image section that is being loaded to block's shared memory

The kernel function that processes individual pixels can be illustrated by the following pseudo code.

```
Calculate from block and thread Ids
ImageCoord(iX, iY)
SharedCoord (bX, bY)

Copy pixel value to block shared memory (Image-
Shared)

if (isBlockBorder(iX, iY)) then
   Load Neighboring Values
end if
syncthreads()
OutputImage[iX,iY] = BooleanEdgeDetection( As,
bX, bY, offset)
syncthreads()
```

The first step to calculate the image pixel coordinates, and appropriate indexes for the stored shared memory. The next step is to copy the pixel value from global memory to shared memory, and in case it is a block border, pixel and its neighbors. After this it is necessary to synchronize all the threads, which means making sure all the data has been copied to the shared memory. Now it is possible to conduct the arithmetic operations for checking edge existence using function $BooleanEdgeDetection$, and storing them in a global output array. Finally, before exiting the kernel thread, synchronization needs to be done.

Function $BooleanEdgeDetection$, calculates edge existence for individual pixels. First the 3x3 window is converted into a bit map pattern as ex-

ISBN: 978-1-61804-089-3

plained in Section 2. A bit patterns is stored as an integer value in a form of a bit mask containing 9 flags. Each flag stands for one position in the 3x3 matrix. The acquired bit mask is compared to predefined bit masks that correspond to the 16 possible edge patterns. The use of flags has the advantage of lowering the need for memory of individual threads. $BooleanEdgeDetection$ also implements the removal of false edges using the global threshold. We can see in Figure 3 an example of edges detection using this approach.

Figure 3: Top: Lena Image, Bottom: Detected edges using the Local Threshold and Boolean Function Based Edge Detection Algorithm

Table 1: Comparison of execution time for a 100 repetitions of sequential and parallel CUDA implementation of the Local Threshold and Boolean Function Based Edge Detection Algorithm

Dimensions	Sequential	CUDA	Improvement
128x128	0.078	0.0083	9.40
256x256	0.309	0.0312	9.90
512x512	1.452	0.128	11.34
1024x1024	5.968	0.528	11.30
2048x2048	20.535	1.613	12.72

4 Tests and Results

In this section we compare the performance of the sequential and proposed CUDA implementation of the edge detection algorithm. Both algorithms are implemented using Microsoft Visual Studio 2010 combined with CUDA version 4.0 for the parallel implementation. The calculations have been done on a machine with Intel(R) Core(TM) i7-2630 QM CPU @ 2.00 GHz, 4GB of DDR3-1333 RAM, with Nvidia GTX 540M 1GB graphics card running on Microsoft Windows 7 Home Premium 64-bit). The graphics card had 96 CUDA Cores.

In the parallel algorithm the size of a thread block was 32x32. In article [10], the effectiveness of CUDA is proven by comparing the authors parallel algorithm to the Intel Open Computer Vision Performance Library (OpenCV) that contains a CPU assembly optimized version of the Canny detector, capable of multithreaded multi-core operation. In our tests, we wish to show the level of improvement that using CUDA gives compared to the sequential algorithm.

We have done our tests on several images of different sizes, ranging from 128x128 to 2048x2048. For each image size we observe the calculation time necessary for finding edges in a 100 repetitions. A single execution of edge detection is very fast, and without the repetitions the results would possibly be prone to outside factors. We present our results in Table 1.

The time presented in Table 1, is given for the processing time excluding the time for loading the bitmap image into the computer memory. In case of the CUDA implementation the allocation of device memory, copying of the data to and from the device memory is included in the calculation time. We can see that the speedup of the CUDA implementation is significant and is around ten times faster than the sequential algorithm. The improvement slightly grow with the increase of the image size, from 9.40 in the case of 128x128, up to 12.72 in case of 2048x2048 image.

ISBN: 978-1-61804-089-3

This great improvement justifies the use of CUDA for edge detection and makes it possible to conduct this type of image processing on images of large dimensions.

5 Conclusion

We have presented a parallel implementation of the Local Threshold and Boolean Function Based Edge Detection Algorithm using CUDA. We have shown that the use of the GPU can greatly improve the speed of these calculations compared to sequential calculations on the CPU. In our tests it was a significant speedup of ten times.

We should emphasize that the method used here could be made more efficient, and it should be possible to further decrease execution time using a more sophisticated parallel algorithms exploiting the texture memory space available in CUDA. Our experience shows that using CUDA can move even more complex image processing algorithms to the GPU.

References:

[1] E. Nadernejad, S. Sharifzadeh and H. Hassanpour, Edge Detection Techniques: Evaluations and Comparisons, *Applied Mathematical Sciences*, Vol. 2, 2008, 2–31,pp. 1507–1520

[2] LS. Davis, A survey of edge detection techniques, *Computer Graphics and Image Processing*, Vol. 4, No. 3 1975, pp. 248–260

[3] J. Canny, A Computational Approach To Edge Detection, *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1986, 8–6, pp. 679-698.

[4] M.B. Ahmad, T.S. Choi, Local Threshold and Boolean Function Based Edge Detection, *IEEE Transactions on Consumer Electronics*, 1999, 45–3, pp. 674-679.

[5] Furferi R., Governi L., Palai M., Volpe Y., Artificial vision based inspection of marbled fabric, *Proceedings of the 5th International Conference on Computer Engineering and Applications CEA'11*, 2011, pp. 93-98.

[6] Yuan-Hui Yua, Chin-Chen Chang, A new edge detection approach based on image context analysis, *Image and Vision Computing*, Vol. 24, Issue 10, 2006, pp. 10901102.

[7] NVIDIA CUDA C Programming Guide Version 4.0, 2011.

[8] P.O. Jaaskelainen, C.S. de La Lama, P. Huerta, and J.H. Takala, , OpenCL-based design methodology for application-specific processors, *Embedded Computer Systems (SAMOS), 2010 International Conference on* , 2010, pp. 223–230.

[9] Direct Compute Lecture Series, http://channel9.msdn.com/tags/DirectCompute-Lecture-Series/, 2011

[10] L. Yuancheng and R. Duraiswami, Canny edge detection on NVIDIA CUDA, *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, 2008, pp. 1–8.

[11] Q. Wu, Z. Fu, C. Tong and Q. Wang, The method of parallel Gabor wavelet transform edge detection based on CUDA, *Environmental Science and Information Application Technology (ESIAT), 2010 International Conference on*, 2010, pp. 537–540 .