Original Article

A New Parallel Approach for Accelerating the GPU-Based Execution of Edge Detection Algorithms

Zahra Emrani, Soroosh Bateni[1], Hossein Rabbani

Medical Image and Signal Processing Research Center, Isfahan University of Medical Sciences, [1]Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran

Abstract

Real-time image processing is used in a wide variety of applications like those in medical care and industrial processes. This technique in medical care has the ability to display important patient information graphi graphically, which can supplement and help the treatment process. Medical decisions made based on real-time images are more accurate and reliable. According to the recent researches, graphic processing unit (GPU) programming is a useful method for improving the speed and quality of medical image processing and is one of the ways of real-time image processing. Edge detection is an early stage in most of the image processing methods for the extraction of features and object segments from a raw image. The Canny method, Sobel and Prewitt filters, and the Roberts' Cross technique are some examples of edge detection algorithms that are widely used in image processing and machine vision. In this work, these algorithms are implemented using the Compute Unified Device Architecture (CUDA), Open Source Computer Vision (OpenCV), and Matrix Laboratory (MATLAB) platforms. An existing parallel method for Canny approach has been modified further to run in a fully parallel manner. This has been achieved by replacing the breadth-first search procedure with a parallel method. These algorithms have been compared by testing them on a database of optical coherence tomography images. The comparison of results shows that the proposed implementation of the Canny method on GPU using the CUDA platform improves the speed of execution by 2–100× compared to the central processing unit-based implementation using the OpenCV and MATLAB platforms.

Keywords: *Algorithms, computer systems, computers, humans computer-assisted image processing, optical coherence tomography*

Introduction

In the past, graphic processing units (GPUs) have been used to execute only graphic applications, and implementing parallel processing algorithms on this platform was extremely challenging. In recent years, GPUs have progressed into general purpose graphics processing units (GPGPUs) and have been perfect as a source of parallel computing.[1] GPGPUs are comparable with field-programmable gate arrays (FPGAs) in power consumption and Gflops performance. The FPGA and GPGPUs have been compared systematically by Cope *et al.*[2] On this basis, FPGA is more appropriate for algorithms with large numbers of memory access, whereas GPGPUs are suitable for algorithms with variable data.

Compute Unified Device Architecture (CUDA) is a software which can be used to obtain better parallelism in a GPGPU using single instruction multiple threads. As a result, developing GPGPU programs becomes more efficient. Nevertheless, implementing image processing algorithms on GPGPUs in parallel is still challenging because organizing thread and memory hierarchy have a great effect on efficiency.

Many image processing works were performed on GPU before the advent of CUDA and GPGPUs. Fast Fourier transform operations were implemented on GPU by Moreland and Angel.[3] They gained a 4× speedup compared to the generic implementation on central processing unit (CPU).

Strzodka and Garbe[4] implemented a motion estimation algorithm on both the GPU and CPU. In comparison, the GPU-based execution was 2.8× faster. Color space conversion for Moving Picture Experts Group (MPEG) video encoding was implemented by Shen *et al.*[5] on GPU using the DirectX, which improved the implementation speed 2–3×.

How to cite this article: Emrani Z, Bateni S, Rabbani H. A new parallel approach for accelerating the GPU-based execution of edge detection algorithms. J Med Sign Sence 2017;7:33-42.

Address for correspondence: *Mrs. Zahra Emrani, Medical Image and Signal Processing Research Center, Isfahan University of Medical Sciences, Isfahan 81745-319, Iran.*
E-mail: zahra_emrani@mail.mui.ac.ir
Both authors 1 and 2 have contributed equally.

Website: www.jmss.mui.ac.ir

Many parallel algorithms developed for computer vision have been presented.[6]

The GPU is now considered a general-purpose platform and is easily programmable. Hence, some open-source libraries such as OpenVIDIA[7] are now available for improving the GPU-based algorithms. Another open-source library called GpuCV[8] provides a platform for connecting the GPU and Open Source Computer Vision (OpenCV). MinGPU[9] is another open-source library that presents a set of practical algorithms for image processing and computer vision.

Image processing techniques such as registration, classification, and feature extraction are important topics in medical applications. GPU programming is a useful method for improving the speed and quality of these procedures, especially in the field of medical image processing.[10-16]

Until now, several algorithms have been introduced for edge detection on GPU.[17-22] Luo and Duraiswami[23] implemented the Canny edge detection method using the CUDA platform. This implementation uses the breadth-first search (BFS) for tracing the edges and for hysteresis thresholding. This method is time-consuming and inefficient. In our method, we offer a solution to eliminate the need to use the BFS. The other parts of the Canny method are fairly straightforward and highly parallel. Moreover, all the well-known edge detectors are combined and presented as a real-time interface, which are implemented in both CUDA and OpenCV. It is also possible to easily modify the different inputs and variables. The results are compared with Matrix Laboratory (MATLAB) and other OpenCV and CUDA implementations.

Graphic processing unit

Physical constraints and high power consumption are the most crucial factors in limiting the use of CPU. By increasing the number of processing cores, the performance improves; however, power consumption goes up as well. In comparison, a GPU consumes less power and has many more processing cores than a CPU. However, the GPU processing cores are much more basic than the CPU cores. The architectures of CPU and GPU are illustrated in Figure 1.

Compute Unified Device Architecture

CUDA is a parallel computing framework and programming platform introduced by NVIDIA.[6] One of the best advantages of using parallel programming on GPU is the reduction of runtime for massively parallel algorithms.

Figure 1: Architectures of CPU and GPU[6]

By taking advantage of the computational ability of GPUs, researchers can achieve a higher performance and reduce the analysis time from several minutes to a few seconds.

Memory management is an essential part of CUDA programming. CUDA framework defines five types of memory: global memory, texture memory, shared memory, registers, and local memory. Global memory has a large capacity. However, this memory is off-chip and it is not cached; therefore, there is a large delay in accessing this memory. Texture memory and constant memory are also located in the global memory; nevertheless, they are cached memories. Accessing the constant and texture memories is fast, but these memories are read-only. Shared memory and the registers are on-chip, but their size is much smaller compared to the global memory. Registers are the fastest memory available in CUDA framework. Registers have a much smaller capacity (in the order of kilobytes) relative to the global memory. Shared memory has an access time which is as fast as that of registers, but it also has a similar limited capacity. Shared memory has a shared storage space with the on-chip cache memory.

Three important terminologies exist in CUDA: thread, warp, and thread block. Thread is the smallest component of parallel computing. A thread processes a single data from a stream of data, for example, a pixel of an image. A block is an arbitrary group of threads. Shared memory is only accessible from within a thread block. The execution of 32 parallel threads is called a warp. Each warp is executed independently. Using a single data for each warp allows the threads in the same warp to run in parallel.

The rest of the paper has been organized as follows. The second section reviews the famous edge detection algorithms. Our method of implementing these edge detection algorithms on GPU is explained in the third section. Experimental results are reported and discussed in the fourth section. Finally, the conclusion of this study is provided in the final section

Edge Detection Algorithms

Feature extraction is one of the early stages in image processing, and edges are usually the most frequently extracted features. There are many methods available for the detection of edges in a given image. The Canny method[24] is the most famous edge detection technique and is widely used. Using this method for certain kinds of images (i.e., images that are usually smoothed by a Gaussian filter) yields a better result compared to other algorithms such as Sobel filter,[25,26] Prewitt filter,[27] and Roberts' Cross.[28]

An edge delineates a local change of intensity in an image. Edges usually occur at the boundary between two different regions of an image.

The edge detection goals can be summarized as follows:[29,30]

- Providing a line drawing of a scene from an image of that scene.
- Extracting the edges of an image, such as corners, lines, and curves.
- Providing inputs for vision algorithms.

To implement an edge detection algorithm, four general steps must be followed:[29,30]

(1) Smoothing: Removes noises as much as possible without any effect on the edges.
(2) Enhancement: Executes a filter for edge enhancement.
(3) Detection: Determines which edge should be removed and which edge should be maintained. (Typically, thresholding is used for this step.)
(4) Localization: Specifies the particular position of an edge.

General steps of an edge detector algorithm

Figure 2 illustrates the main steps in the edge detection procedure. Each algorithm is described in the following sections.

Roberts' Cross

The Roberts' Cross edge detector has been proposed by Lawrence Roberts in 1965.[28] In brief, this detector is a discrete differentiation operator that calculates the horizontal and vertical changes in the input. In this way, the original image is convolved with two 2×2 kernels.

Eq. (1) shows this operator:

$$\begin{aligned}\frac{\partial f}{\partial x} &= f(i,j) - f(i+1,j+1)\\ \frac{\partial f}{\partial y} &= f(i+1,j) - f(i,j+1)\end{aligned} \quad (1)$$

The two previous kernels could be replaced by these approximations:

$$M_x = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}, \quad M_y = \begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix} \quad (2)$$

Consequently, the horizontal and vertical changes can be calculated by Eq. (3).

$$F_x = M_x * F, \quad F_y = M_y * F \quad (3)$$

F is the original image, M_x and M_y are the two kernels, and F_x and F_y are the images that show the horizontal and vertical changes. The "*" indicates the convolution, which will be discussed in "Materials and Methods" section.

Prewitt filter

Judith M.S. Prewitt proposed the Prewitt filter in 1970.[7] In this algorithm, the original image is convolved with two 3×3 kernels. These kernels are represented by F_x and F_y in Eq. (4). This algorithm is similar to the Roberts' Cross algorithm.

$$F_x = \begin{bmatrix}-1 & 0 & +1\\ -1 & 0 & +1\\ -1 & 0 & +1\end{bmatrix} * F, \quad F_y = \begin{bmatrix}+1 & +1 & +1\\ 0 & 0 & 0\\ -1 & -1 & -1\end{bmatrix} * F \quad (4)$$

• Main steps in edge detection using masks

(1) Smooth the input image ($\hat{f}(x,y) = f(x,y) * G(x,y)$)

(2) $\hat{f}_x = \hat{f}(x,y) * M_x(x,y)$

(3) $\hat{f}_y = \hat{f}(x,y) * M_y(x,y)$

(4) $magn(x,y) = |\hat{f}_x| + |\hat{f}_y|$

(5) $dir(x,y) = \tan^{-1}(\hat{f}_y/\hat{f}_x)$

(6) If $magn(x,y) > T$, then possible edge point

Figure 2: Main steps for Roberts' Cross, Prewitt, and Sobel edge detectors[29,30]

Sobel filter

Sobel proposed the Sobel filter.[4-6] This detector is similar to the Prewitt edge detector, but with different kernels. Eq. (5) expresses these two kernels.

$$F_x = \begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix} * F, \quad F_y = \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ +1 & +2 & +1\end{bmatrix} * F \quad (5)$$

Canny edge detector

The Canny algorithm[24] is the most famous edge detector used in image processing. Applying this algorithm on certain kinds of images (e.g., images containing Gaussian noise) yields better results compared to other algorithms. The main steps in the Canny algorithm are illustrated in Figure 3.

Materials and Methods

The methods used in this paper are based on GPU4Vision.[6] As was discussed in "Edge Detection Algorithms" section, each edge detection algorithm includes at least one convolution step, which generally can be expressed as

$$F_{x|y} = M_{x|y} * F \quad (6)$$

F is a subset of the original matrix (here representing a part of an image). The type of algorithm used defines the size of F. $M_{x|y}$ represents the kernels in each algorithm (M_x or M_y, depending on the direction), and $F_{x|y}$ is an integer value. $F_{x|y}$

Algorithm

1. Compute f_x and f_y

$$f_x = \frac{\partial}{\partial x}(f * G) = f * \frac{\partial}{\partial x} G = f * G_x$$

$$f_y = \frac{\partial}{\partial y}(f * G) = f * \frac{\partial}{\partial y} G = f * G_y$$

$G(x,y)$ is the Gaussian function

$G_x(x,y)$ is the derivate of $G(x,y)$ with respect to x: $G_x(x,y) = \frac{-x}{\sigma^2} G(x,y)$

$G_y(x,y)$ is the derivate of $G(x,y)$ with respect to y: $G_y(x,y) = \frac{-y}{\sigma^2} G(x,y)$

2. Compute the gradient magnitude

$$magn(i,j) = \sqrt{f_x^2 + f_y^2}$$

3. Apply non-maxima suppression.

4. Apply hysteresis thresholding/edge linking.

Figure 3: Main steps in Canny edge detection procedure[29,30]. Illustration of the local gradient direction and its effect on the output (the calculated direction is horizontal); (a) The gradient direction is vertical, (b) The gradient direction is horizontal

is calculated vertically (F_y) or horizontally (F_x) depending on the direction of M. The "*" denotes a convolution between two matrices and is defined as follows:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2] \quad (7)$$

Moreover, the method proposed by Canny involves another convolution to smooth the input:

$$F = F * G \quad (8)$$

G is the Gaussian function and is further explained in "Parallel convolution" section. The smoothing is the first step in the Canny approach. The output (F) would be the input (F) in Eq. (6). The final value of a pixel is calculated by Eq. (9):

$$F_{out(i,j)} = \sqrt{F_x^{\,2} + F_y^{\,2}} \quad (9)$$

Eq. (9) specifies the geometric distance of F_x and F_y from the center of the image and is called the gradient magnitude. F_{out} is the output image. (i, j) denotes the index of a pixel in the output image. For a 3×3 kernel, (i, j) corresponds to the center of the input matrix (F) in Eq. (6).

Except for the Gaussian filter, the other initial steps of the Canny method are common with the other edge detection algorithms and could be efficiently parallelized. However, as was explained earlier, Canny has proposed two additional steps to enhance the output. These steps have been parallelized, but they could be improved further. First, the efficient method of parallel convolution (the method used in the initial steps of the Canny method) is discussed in "Parallel convolution" section. Afterwards, the two additional steps of the Canny approach are described in "Non-Maximum suppression" and "Edge tracing and hysteresis thresholding" sections, and a method is proposed to further enhance the parallelization.

Parallel convolution

As was mentioned above, an essential part of the process of parallelizing edge detection algorithms is to have an efficient parallel convolution.

To efficiently parallelize the convolution [Eq. (6)], several steps should be taken. First, M_{xly} and F should be localized for each kernel. Usually, M_{xly} is a 2×2 or 3×3 matrix and is fixed for each algorithm. Therefore, it is generally a good idea to store this matrix in the shared memory of each streaming multiprocessor (SM), because the shared memory has fewer fetch cycles compared to other types of memory. Moreover, to calculate each new pixel for the output image, each algorithm only needs a small subset of the input image. As an example, the Sobel filter needs eight adjacent neighbors of each pixel, as is shown in Figure 4. Since the number of pixels that are needed for each new pixel is generally small (nine for Sobel), this set could easily be stored in the cache of each SM. Storing both the F and M_{xly} matrix in the shared memory vastly enhances the memory access delay.

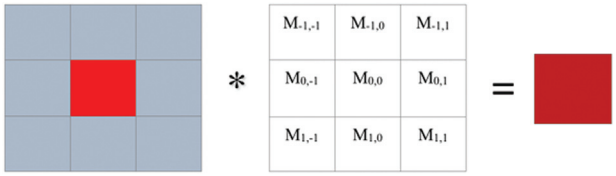

Figure 4: Outcome of matrix multiplication used in Sobel filter

Now, we have demonstrated the two essential steps for efficiently parallelizing a convolution. However, there are further steps that could be taken to have a better parallel algorithm. These set of "hacks" are specific to each algorithm.

For the Canny edge detector, the Gaussian matrix does not theoretically have a size limitation. Thus, the Gaussian matrix could be applied to any portion of the input image. However, since the Gaussian function used to generate the Gaussian matrix decays rapidly, it is reasonable to make the window size smaller.

In the proposed method, a 3×3 Gaussian filter is used to both improve the accuracy and simplify the calculations. Any small dimension would satisfy the accuracy requirement. The 3×3 dimension is deliberately chosen since the Sobel filter used in the Canny method also uses a set of 3×3 kernels. Since only one device call is needed to compute both the Gaussian filter and Sobel filter, this selection would also be more appropriate from a computational perspective. As a note, a 5×5 size for the G matrix is also fine, but it has one of the following drawbacks. The first drawback is the unnecessary local memory usage. This is due to the fact that the Sobel filter only uses a 3×3 matrix. Another drawback is the unnecessary device kernel call. Moreover, for the sake of efficiency, since the size of the Gaussian matrix is small, it is more efficient to store it individually for each SM in the shared memory. However, in image processing's point of view, the best size of window should be obtained according to the resolution and type of image (such as crowded images, texture images, and smooth images), level of noise, etc.[31]

We can further simplify our algorithms. A good step is to unroll the kernels of all proposed algorithms to eliminate the burden of computing a matrix dot product each time a device kernel is called. Since the kernel sizes are different for each method, we have to divide our algorithms to two subgroups:

Eq. (7) could be unrolled as follows for both 3×3 M_{xly}. These kernels are used in Sobel filter, Canny, and Prewitt filter. Specifically, we are using the same size for our Gaussian

matrix too.

$$F_{x|y} = \Big(M_{x|y}[-1][-1]\cdot F_s[1][1] + M_{x|y}[0][-1]\cdot F_s[0][0] + \cdots + M_{x|y}[1][1]\cdot F_s[-1][-1]\Big) \quad (10)$$

And for 2×2 matrices used in Roberts' Cross, it is unrolled as

$$F_{x|y} = \Big(M_{x|y}[-1][-1]\cdot F_s[0][0] + M_{x|y}[0][-1]\cdot F_s[-1][0] + M_{x|y}[-1][0]\cdot F_s[0][-1] + M_{x|y}[0][0]\cdot F_s[-1][-1]\Big) \quad (11)$$

In Eqs. (10) and (11), F_{xIy} indicates a pixel in the output image. It is intuitively concluded from Eqs. (10) and (11) that all F_{xIy} values are independent of each other. Hence, as was mentioned earlier, by storing M_{xIy} and F values in the local memory, the calculations of the output pixels could be run in a fully parallel manner.

On the other hand, the kernel that is calculating the output pixel would need to fetch the F from the global memory the first time it needs to access it. In addition, as is apparent in the equations, some data are redundant between each device kernel. This creates a race condition between device kernels. In such a case, all the conflicting accesses to the global memory would be serialized. To avoid this condition, the texture memory is used to fetch the F matrix from the global memory. The texture memory incorporates several steps of caching that, unlike the other cache memories on GPU, contains a locality function. If a single address is attached to the texture memory and it is fetched, several neighboring memory locations would also be fetched and stored in the local cache. In our case, this functionality is very useful. The input image does not change throughout the kernel call and it needs the locality function that the texture memory offers to avoid the race condition.

Except for Canny, calculating the geometric distance of each pixel is the final step for all the famous algorithms discussed in this paper [Eq. (9)]. For each of these algorithms, a separate kernel is programmed and called.

Moreover, as was mentioned earlier, the Canny method uses the Sobel filter to extract the edges. However, to enhance the results, the input is first convoluted with a Gaussian filter. For this purpose, two kernels are launched. One of them computes the G matrix and the other performs a parallel convolution on the input image. In addition, exceptionally in the case of Canny, the direction of each edge should be known for the future steps. Hence, our deployed Sobel filter device kernel must be modified specifically for this need. The direction of each edge is individually stored in an array called θ. The grid size for this kernel is set equal to the number of pixels.

In addition, in the case of Canny, two more steps need to be executed on the output image before it could be ready. These two steps are described in the following sections.

Non-maximum suppression

As was mentioned in previous sections, other proposed algorithms are now at their most efficient levels. However, the Canny method needs further attention since it includes two more steps in the end to further enhance the results.

After applying the Gaussian and Sobel filters on the input image, the output would contain the extracted edges. However, these edges include noise and broken edges, which is not desirable. In non-maximum suppression, the so-called "shadows" are removed from the extracted edges in an effort to reduce the noise. The shadows are formed around the actual edges and, therefore, make them thicker than they really are. To remove the shadows from the edges, it is assumed that a thick edge is made of several thinner edges with different intensities. Since the calculations are done at pixel level and there are only eight neighbors to each pixel, the directions of these pixels are limited to four: horizontal, vertical, +45°, and −45°. The calculated directions in the previous step are used here and are classified as one of the above four groups.

Among the parallel lines, the line with the highest intensity is considered the actual edge and the other lines are removed (set to zero). This process for two cases is illustrated in Figure 5. The abovementioned method results in edges with a thickness of only one pixel. This is due to the fact that only one pixel is chosen from a set of local pixels. This method might also result in distorted edges, since the highest intensity does not always belong to the most desired pixel, and it might belong to a shadow. Nevertheless, this step is intuitively parallel, since the outcome of each pixel does not depend on the results of neighboring pixels. The highest gradient magnitude, which is chosen locally, is perfect for the GPU architecture. There is no need to prevent the neighboring pixels results written to memory from affecting the current kernel calculations, since the highest level of gradient is chosen in each kernel.

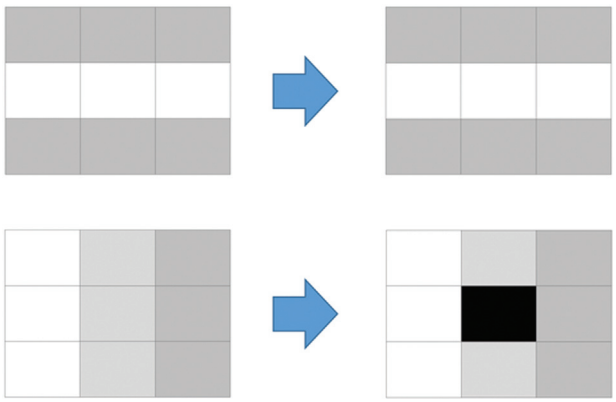

Figure 5: Illustration of the local gradient direction and its effect on the output (the calculated direction is horizontal). (a) The gradient direction is vertical. (b) The gradient direction is horizontal

Edge tracing and hysteresis thresholding

The final step in the Canny approach is tracing the edges. In the previous steps of this method, edges were extracted and shadows were removed. A hysteresis threshold is used for detecting the corrupted edges. A corrupted edge is defined as a false edge or an absence of pixels in a true edge.

Two threshold bounds are considered: a high threshold and a low threshold. Each pixel with a value above the high threshold is considered a strong edge. Likewise, each pixel with a value below the low threshold is considered a noisy pixel and is removed from the result. Pixels whose values fall in between these two thresholds are considered potential edges (weak edges). Every weak edge should be evaluated and designated as either a strong edge or noise until there are no more weak edges. A separate kernel is executed for this step. This kernel could not be combined with the previous kernel due to the lack of global synchronization in GPU. This kernel needs a fixed value for all the pixels to run in a fully parallel manner. Therefore, when the previous kernel has fully calculated the pixel values, this kernel would be called from the host. The kernel checks whether a pixel is a potential edge or not. If so, it will check its neighboring pixels. If a strong edge is found, the current pixel would be considered as an edge pixel. However, if no strong edge is found, no decision could be made at this point.

A global flag residing in the global memory indicates whether any pixel has changed its status from a potential edge to a strong edge or not. If a pixel has changed its status, the kernel should be re-launched to examine the eight neighboring pixels for potential weak edges. It is inefficient to localize a kernel to a single neighborhood. Moreover, managing the memory in this way is hard, if not impossible. Therefore, the authors have chosen to re-launch the kernel on the whole image. The future works can focus on a potential way of improving the efficiency by localizing the kernel launch to the changed pixels' neighborhoods. In the proposed method, the abovementioned kernel is launched as long as the flag is set.

This method eliminates the need to run a BFS algorithm on GPU. BFS[32-34] is an algorithm for finding the shortest path in graph searches by starting at the tree root, exploring the neighbor nodes, and then moving to the neighbors of next level for exploring.

Running the BFS algorithm in parallel has proved to be inefficient. There might be conflicting pixels from each starting point. Unless the shared memory is used, access to memory locations in the conflicting zones would be serialized. The test results also indicate a boost in performance. Executing the heavy and unnecessary BFS algorithm is much slower than running the kernel in the manner proposed. Nevertheless, the quality of the final outcome suffers a little due to the loss of precision that the BFS algorithm offers. Our test results show that the degradation in quality is negligible.

The global flag will generate a race condition. If a thread changes a pixel's status, it will try to set the flag. This could generate a queue of memory write requests as big as the input size. One way of avoiding this situation to some extent is to see whether the flag is already set or not. In this manner, each warp (32 threads) will read the memory location only once and will not request a write to memory if the flag is already set. Hence, the queue size would be reduced. Our test results show that the overhead of the flag race condition is also negligible compared to the requirements of running the BFS algorithm.

Eventually, the edges that are not recognized as strong edges and also those that do not have any strong neighboring pixels are removed from the result. For this purpose, a kernel is executed on the result, and it assigns a value of zero to each pixel that is not considered a part of a strong edge.

Results

These algorithms have been tested using a desktop personal computer. Table 1 shows the detailed configuration of this system, and the specifications of our GPU are listed in Table 2.

The proposed method is tested on some randomly selected two-dimensional enhanced depth imaging optical coherence tomography (EDI-OCT) images obtained from 10 eyes of six normal (B-scan (512×768)×61 slice) subjects by an EDI system of multimodality diagnostic imaging (wavelength: 870 nm; scan pattern: EDI; Spectralis HRA + OCT; Heidelberg Engineering, Heidelberg, Germany).[35,36]

Experimental results

The relationships between image size and execution time on CPU (using the OpenCV standard functions) and GPU obtained by applying the four abovementioned algorithms

Table 1: Basic system configuration

Host CPU	Intel(R) Core(TM) i7 CPU 3930 K (3.2 GHz)
GPU	NVIDIA Geforce GTX 550 Ti
CUDA version	5

Table 2: GPU specifications

Architecture	GTX 550 Ti
# of CUDA cores	192
# of transistors (millions)	1170
# of SMs	4
Graphic clock (MHz)	900
Processor clock (MHz)	1800
Memory clock (Gbps)	4.1
Standard memory configuration	1024
# of texture units	32
Processing power (GFLOPS)	691.2
Memory capacity (GB)	GDDR5 (1 GB)

is presented in Figure 6. The corresponding speedup graph is also shown in Figure 6. This figure shows that the execution time on GPU is less than that on CPU. Unless the image size is too large for its storage in the cache, the CPU can handle these algorithms efficiently. On the other hand, on the GPU, the large number of parameters might quickly lead to performance degradation, because the threads are grouped into blocks, and it is hard for different blocks to communicate with each other. Furthermore, the cache in each block is local and cannot be accessed by threads from other blocks. Nevertheless, despite the massive number of threads, the GPU can achieve a higher performance by hiding the memory access latency.

In addition, Figure 6 shows that the Canny edge detector can achieve a speedup of up to 100×, whereas the Sobel algorithm can speed up the execution time by up to 400×. The speedup curves in Figure 6 vary and depend on the image size.

Performance analysis

Unfortunately, there is no standard way of comparing the powers of CPU and GPU. A good measure is floating point operations per second (FLOPS). However, it is unfair to compare a CPU and a GPU with the same FLOPS. For one thing, theoretically, our CPU (being a decent CPU) is able to execute 38.4 GFLOPS (12 × 3.2 to 12 × 4 with turbo boost technology). However, in real world, it is usually less than that. On the other hand, our GPU, at nearly half the price, performs 62.34 GFLOPS for double precision operations, nearly 2× the theoretical performance of our CPU.

In addition to the above explanation, there are many factors that can affect the performance of a GPU, input size being one of the important factors. Table 3 shows the comparison between the results obtained for the same picture with different sizes. The size of the input image has a direct effect on the texture memory used, and it also influences the usage of cache and registers per block. The input image size has a noticeable effect on the amount of memory used in both the host and device. For statistical purposes, the standard deviation along with the calculated average of the runtime for Canny method using the 512 × 512 input is given in Table 4. Table 3 shows the variation in runtime.

Another factor that affects the performance in Canny method is the threshold. Different thresholds result in different runtimes, since different levels of examination have to be implemented on the input. Particularly on the GPU, the size of the block and the number of blocks per grid directly affect the execution of the threads. In the Fermi architecture, each SM is limited to run only eight blocks in parallel; any more blocks assigned to an SM would be queued.

In addition, on our device, a single SM is limited to run 2048 threads in total and a maximum of 1024 threads per block. Depending on the block size and the number of blocks in total, different numbers of SMs are used for a particular application. For example, our GPU has four SMs. With a configuration of eight blocks with 256 threads each, only one SM is used. However, when using 16 blocks with 128 threads each, two SMs are used, which results in a much faster execution. This happens because the blocks running on a single SM share the same resources (cache and register). For example, if a SM runs out of cache, some blocks will not launch and therefore will be queued.

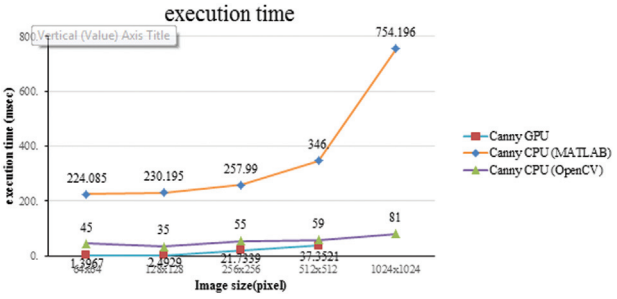

Figure 6: Comparison between execution times in CPU and GPU

Table 3: Running times of GPU (CUDA) on OCT image of retina

Algorithms	Image size				
	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024
Canny	1.3967 (ms)	2.4928 (ms)	21.7339 (ms)	37.3520 (ms)	187.7202 (ms)
Sobel	0.0673 (ms)	0.2361 (ms)	0.9170 (ms)	3.6284 (ms)	14.7420 (ms)
Prewitt	0.0674 (ms)	0.2353 (ms)	0.9176 (ms)	3.4249 (ms)	14.7376 (ms)
Roberts' Cross	0.0335 (ms)	0.1104 (ms)	0.4144 (ms)	1.5411 (ms)	6.7952 (ms)

Table 4: Standard deviation and average of running times of GPU (CUDA) on OCT image of retina (512 × 512) for Canny

Algorithm	Run no.						AVG	SD
	1	2	3	4	5	6		
Canny	41.25 (ms)	36.12 (ms)	29.43 (ms)	47.96 (ms)	31.25 (ms)	38.1 (ms)	37.352 (ms)	6.189 (ms)

Different configurations have been tested. The best result is achieved by setting the horizontal dimension of the input equal to the number of threads per block and the vertical dimension equal to the number of blocks per grid. Since each pixel is convoluted separately, if the resources are available, it will be a good idea to assign each thread to a single pixel.

Another decisive factor is the memory access. To generate the final result, each thread needs a matrix of pixels that are considered neighbors to its pixels. This access is coalesced with the neighboring threads. Using the shared memory or the constant memory would help reduce this memory coalescing in only one direction. To solve this problem in 2D, a 2D texture is used. This results in a 2D local cache that vastly enhances the memory access.

The running times of algorithms depend on many different parameters including the image resolution, the number of objects in the scene, and also the thresholds in Sobel and Canny algorithms. Figures 7 and 8 show examples of "GPU

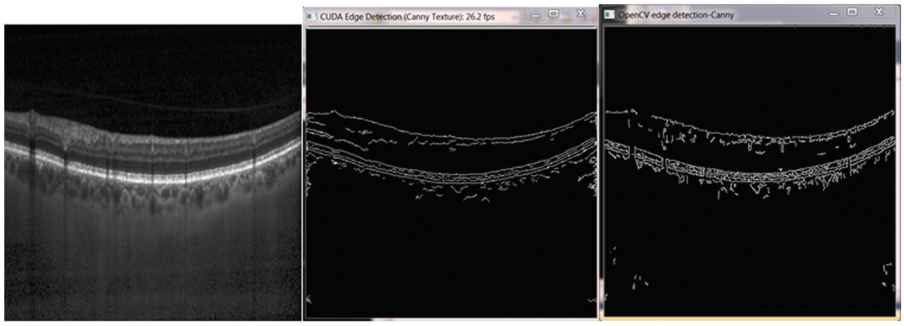

Figure 7: Different implementations of the Canny edge detection method on OCT image

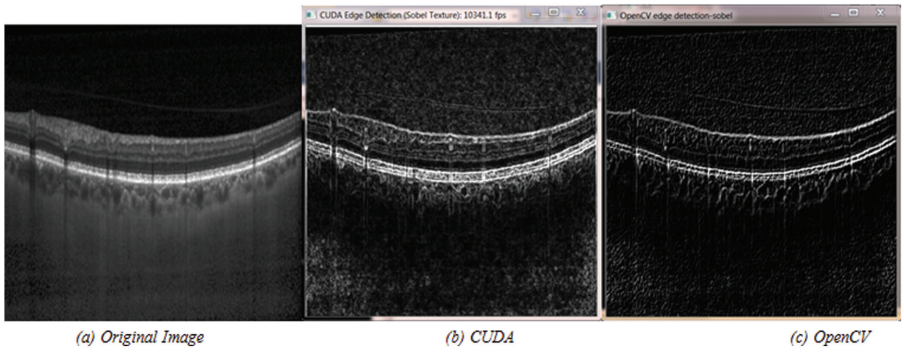

Figure 8: Different implementations of the Sobel edge detection method on OCT image

Table 5: Running times of CPU (MATLAB) on OCT image of retina

Algorithms	Image size				
	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024
Canny	224.085 (ms)	230.195 (ms)	257.99 (ms)	346 (ms)	754.196 (ms)
Sobel	155.237 (ms)	190.751 (ms)	341.331 (ms)	938.582 (ms)	3425.117 (ms)
Prewitt	153.775 (ms)	190.514 (ms)	341.533 (ms)	940.144 (ms)	3414.739 (ms)
Roberts' Cross	144.713 (ms)	182.25 (ms)	330.103 (ms)	931.11 (ms)	3385.174 (ms)

Table 6: Running times of CPU (OpenCV) on OCT image of retina

Algorithms	Image size				
	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024
Canny	45 (ms)	35 (ms)	55 (ms)	59 (ms)	81 (ms)
Sobel	50 (ms)	45 (ms)	53 (ms)	45 (ms)	2.19231E+14 (ms)

Table 7: Runtime of each algorithm on GPU in milliseconds. (The input is standard Lena image 512 × 512)

Algorithms	Canny	Sobel	Prewitt	Roberts' Cross
Runtime	33.63 (ms)	3.5 (ms)	3.46 (ms)	1.27 (ms)

Edge Detector" using the above algorithms. Tables 5 and 6 illustrate the results of implementing the mentioned methods on CPU using MATLAB and OpenCV. The comparison of results shows that the proposed implementation of the Canny method on GPU using the CUDA platform improves the speed of execution by 2–100× compared to the CPU-based implementation using the OpenCV and MATLAB platforms.

The runtimes are measured using the shown OCT image. However, for future references, we have measured the runtime for the standard Lena image as well. The results are shown in Table 7.

Conclusion

In this work, edge detector algorithms are described and implemented using the CUDA, MATLAB, and OpenCV platforms and are tested on medical images. The obtained execution times from these methods are compared with each other. Four popular types of edge detectors in image processing (Canny, Sobel, Prewitt, and Roberts' Cross) are presented and the involved parallelization process is modified. In Canny method, the necessity to run a BFS algorithm in each iteration is eliminated.

The comparison of results obtained by running these algorithms in the NVIDIA CUDA, MATLAB, and OpenCV platforms indicates that the parallel implementation on GPU (using the NVIDIA and CUDA) achieves a 2–100× faster execution time, compared to the CPU-based (MATLAB) implementation, and slightly better results compared to the OpenCV results for images with dimensions up to 512 × 512, on a Core i7-Extreme 3.2 GHz desktop computer. For instance, for images with a 512×512 resolution, the implementation of the Canny approach using our proposed method and the CUDA platform respectively gained 9 and 1.5× faster execution times when compared to the MATLAB and OpenCV (CPU-based) implementations. The experimental results indicate that the implementation of the edge detector algorithms using the GPGPU definitely leads to practical performance enhancement.

Future works

In the current state of the GPU Edge Detector, the focus is on completeness and the real-time behavior. However, as the results show above, there is room for many enhancements in terms of the application runtime.

Acknowledgements

We would like to thank the Microelectronic Technology Strategic Campaign [Strategic Campaign for Microelectronic Technology] (http://micro.isti.ir) for providing the research funding for this project. We also would like to thank Dr. Mahnaz Etehad Tavakol from the Isfahan University of Medical Sciences for her valuable remarks on this paper.

Financial support and sponsorship

This work has been partially funded by the Medical Image and Signal Processing (MISP) Research Center.

Conflicts of interest

There are no conflicts of interest.

References

1. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Skadron K. A performance study of general purpose applications on graphics processors using CUDA. J Parallel Distrib Comput 2008;68:1370-80.
2. Cope B, Cheung PY, Luk W, Howes L. Performance comparison of graphics processors to reconfigurable logic: A case study. IEEE Trans Comput 2010;59:433-48.
3. Moreland K, Angel E. The FFT on a GPU. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003. p. 112-9.
4. Strzodka R, Garbe C. Real-time motion estimation and visualization on graphics cards. Proceedings of the Conference on Visualization'04, 2004. p. 545-52.
5. Shen G, Gao G-P, Li S, Shum H-Y, Zhang Y-Q. Accelerate video decoding with generic GPU. IEEE Trans Circuits Syst Video Technol 2005;15:685-93.
6. http://www.gpu4vision.org.
7. Fung J, Mann S. OpenVIDIA: Parallel GPU computer vision. Proceedings of the 13th Annual ACM International Conference on Multimedia, 2005. p. 849-52.
8. Allusse Y, Horain P, Agarwal A, Saipriyadarshan C. GpuCV: An opensource GPU-accelerated framework for image processing and computer vision. Proceedings of the 16th ACM International Conference on Multimedia, 2008. p. 1089-92.
9. Babenko P, Shah M. MinGPU: A minimum GPU library for computer vision. J Real-Time Image Process 2008;3:255-68.
10. Zhuo Y, Wu X-L, Haldar JP, Hwu W-M, Liang Z-P, Sutton BP. Accelerating iterative field-compensated MR image reconstruction on GPUs. 2010 IEEE International Symposium on Biomedical Imaging: From Nano to Macro, IEEE, 2010. p. 820-3.
11. Yang J, Feng C, Zhao D. A CUDA-based reverse gridding algorithm for MR reconstruction. Magn Reson Imaging 2013;31:313-23.
12. Stone SS, Haldar JP, Tsao SC, Hwu W-M, Sutton BP, Liang Z-P. Accelerating advanced MRI reconstructions on GPUs. J Parallel Distrib Comput 2008;68:1307-18.
13. Kim K, Park S, Hong H, Shin YG. Fast 2D-3D registration using GPU-based preprocessing. Proceedings of 7th International Workshop on Enterprise Networking and Computing in Healthcare Industry, HEALTHCOM'05, 2005. p. 139-43.
14. Hwu W, Nandakumar D, Haldar J, Atkinson IC, Sutton B, Liang Z-P, *et al.* Accelerating MR image reconstruction on GPUs. IEEE International Symposium on Biomedical Imaging: From Nano to Macro, ISBI'09, 2009. p. 1283-86.
15. Huang T-Y, Tang Y-W, Ju S-Y. Accelerating image registration of MRI by GPU-based parallel computation. Magn Reson Imaging 2011;29:712-6.
16. Han X, Hibbard LS, Willcut V. GPU-accelerated, gradient-free MI deformable registration for atlas-based MR brain image segmentation. IEEE Computer Society Conference on Computer

Vision and Pattern Recognition Workshops, CVPR Workshops'09, 2009. p. 141-8.
17. Gong HX, Hao L. Roberts edge detection algorithm based on GPU. J Chem Pharm Res 2014;6:1308-14.
18. Sarkar S, Venugopalan V, Reddy K, Giering M, Ryde J, Jaitly N. Occlusion edge detection in RGB-D frames using deep convolutional networks. CoRRabs/1412.7007, 2014.
19. Chouchene M, Sayadi FE, Said Y, Atri M, Tourki R. Efficient implementation of Sobel edge detection algorithm on CPU, GPU and FPGA. Int J Adv Media Commun 2014;5:105-17.
20. Ogawa K, Ito Y, Nakano K. Efficient Canny edge detection using a GPU. IEEE First International Conference on Networking and Computing (ICNC), Higashi-Hiroshima, 2010. p. 279-80.
21. Niu S, Yang J, Wang S, Chen G. Improvement and parallel implementation of canny edge detection algorithm based on GPU. IEEE 9th International Conference on ASIC (ASICON), October 25-28, 2011. p. 641-4.
22. Roodt Y, Visser W, Clarke W. Image processing on the GPU: Implementing the Canny edge detection algorithm. International Symposium of the Pattern Recognition Association of South Africa, 2007. p. 1-6.
23. Luo Y, Duraiswami R. Canny edge detection on NVIDIA CUDA. IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPRW'08, 2008. p. 1-8.
24. Canny J. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1986. p. 679-98.
25. Sobel I, Feldman G. A 3×3 isotropic gradient operator for image processing. A Talk at the Stanford Artificial Project, 1968. p. 271-2.
26. Sobel I. Camera models and machine perception. DTIC Document, 1970.
27. Prewitt JM. Object Enhancement and Extraction, vol 75 New York: Academic Press 1970.
28. Roberts L. In: Tippet J, editor. Machine Perception of Three Dimensional Solids, Optical and Electro-Optical Information Processing. Cambridge, MA: IT Press; 1965.
29. Trucco E, Verri A. Introductory Techniques for 3-D Computer Vision, vol 93. Englewood Cliffs: Prentice Hall; 1998.
30. Jain R, Kasturi R, Schunck BG. Machine Vision, vol 5 New York: McGraw-Hill 1995.
31. Rabbani H, Gazor S. Image denoising employing local mixture models in sparse domains. IET Image Processing 2010;4:413-28.
32. Skiena S. The Algorithm Design Manual. Springer; 2008. p. 480 doi: 10.1007/978-1-84800-070-4_4
33. Leiserson CE, Schardl TB. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). ACM Symposium on Parallelism in Algorithms and Architectures, 2010.
34. Lee CY. An algorithm for path connections and its applications. IRE Transactions on Electronic Computers; 1961. http://www.heidelbergengineering.com/us/wp-content/uploads/hra2-aquire-the-perfect-image.pdf.
35. Danesh H, Kafieh R, Rabbani H, Hajizadeh F. Segmentation of choroidal boundary in enhanced depth imaging OCTs using a multiresolution texture based modeling in graph cuts. Computational and Mathematical Methods in Medicine, vol 2014, 2014.