Comput Syst Sci & Eng (2020) 1: 21–37

International Journal of
Computer Systems
Science & Engineering

Enhanced GPU-Based Anti-Noise Hybrid Edge Detection Method

Sa'ed Abed*, Mohammed H. Ali† and Mohammad Al-Shayeji‡

Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, P. O. Box 5969 Safat, Kuwait

Today, there is a growing demand for computer vision and image processing in different areas and applications such as military surveillance, and biological and medical imaging. Edge detection is a vital image processing technique used as a pre-processing step in many computer vision algorithms. However, the presence of noise makes the edge detection task more challenging; therefore, an image restoration technique is needed to tackle this obstacle by presenting an adaptive solution. As the complexity of processing is rising due to recent high-definition technologies, the expanse of data attained by the image is increasing dramatically. Thus, increased processing power is needed to speed up the completion of certain tasks. In this paper,we present a parallel implementation of hybrid algorithm-comprised edge detection and image restoration along with other processes using Computed Unified Device Architecture (CUDA) platform, exploiting a Single Instruction Multiple Thread (SIMT) execution model on a Graphical Processing Unit (GPU). The performance of the proposed method is tested and evaluated using well-known images from various applications. We evaluated the computation time in both parallel implementation on the GPU, and sequential execution in the Central Processing Unit (CPU) natively and using Hyper-Threading (HT) implementations. The gained speedup for the naïve approach of the proposed edge detection using GPU under global memory direct access is up to 37 times faster, while the speedup of the native CPU implementation when using shared memory approach is up to 25 times and 1.5 times over HT implementation.

Keywords: Edge Detection; GPU; Image Processing; Noise; Parallel Processing

1. INTRODUCTION

Edge detection in digital image processing is a vital process for many computer vision applications. It consists of common pre-processing steps, such as object tracking, segmentation, audio/video coding, and image enhancement [1]. Edge detection of the image is an important problem that has been studied for more than 60 years [2]. Assuming u_0 is the given image represented as a function $u_0{:}\Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^d$, the problem is to find u_e where $u_e = (A \cdot u_0) \subset \mathbb{R}^2$ such that it represents an edge map image, and A can be considered any operator as stated in Equation (1). Edges of a grayscale image are represented as a sharp change intensity or discontinuity. The interest of edges strengthened on the important information that the image carries off [3]. Different categories of edge detection have been proposed by many researches [4–7], and most of them have been based on applying a first derivative (gradient) on the given image. Other techniques have been proposed [8–10], based on the second derivative (Laplacian). Another category of edge detection technique is proposed [11–16] to overcome the problem of finding edges in images under different circumstances with different methodologies.

$$U_e = \frac{\sum \left(\sum a_{ij} \cdot u_{ij} \right)}{S} \tag{1}$$

where a_{ij} is the coefficients of convolution kernel, u_{ij} is the image data and S is the summation of coefficients if ($|S| > 0$ otherwise $S = 1$).

Ordinarily, images are often mixed with noise, and even a little noise makes the edge detection process challenging. Noise reduction is a well-known problem in digital image processing, which draws from several type of resources, such as transmission medium, imaging system, recording process, or any combination that affects the subjective quality of the edge detection image [17]. An adaptive solution is needed to enhance the accuracy of

*s.abed@ku.edu.kw
†mohammed.cpe@gmail.com
‡m.shayeji@ku.edu.kw

existing detectors by creating a mash-up detector that combines the advantages of existing approaches with a preserving edge noise removal.

Our main contributions in this paper are as follows: 1) we propose an enhanced anti-noise hybrid image edge detection method; 2) we implement the proposed method in the Graphical Processing Unit (GPU) under naïve and shared memory approaches; and 3) we evaluate the proposed solution and prove its efficiency.

The proposed method consists of two phases derived from [18]. The first phase of the proposed algorithm is the pre-processing phase, where the RGB color map image is converted to grayscale image type, using an NTSC weighting approach for further image processing operations. The second phase involves the edge detection process. Here, the result of ETVSB (the overall operator noise detection and removal) is convoluted using a high-pass operator to detect edges in the images. The high-pass operator is an enhanced operator from existing detectors. Ordinarily, a Sobel operator is an excellent operator used in Canny. Besides, Laplacian is an isotropic detector that can detect details in edges like lines and dots. Hence, an adaptive edge detection is proposed to get the benefit from both gradient and Laplacian methods.

Many image processing algorithms including edge detection inherent parallelism. A good selection to speed up the process and exploit the parallelism in the algorithm is to adapt the power of GPU and harness the enormous stream processors. Therefore, a GPU is used to boost the speed of sequential execution as a proof of concept by proposing a parallel implementation of the edge detection approach using a Computed Unified Device Architecture (CUDA) programming model. Several parallel implementations have been conducted in this work, such as thresholding, gradient calculation, maximum image between two image arrays, NMS, and finally edge detection. The edge detection process is implemented using two main approaches. The first approach is to use a native convolution process that directly accesses global memory to warp data, while the second approach uses caching (shared memory) inside each block, rather than working directly with the global memory which is time consuming. The speedup gained from implementing the first approach is up to 25 times, while the latter is up to 37 times and 1.5 times over HT implementation.

A GPU-accelerated road extraction method in images is presented in [19] where the edges are detected by the bi-windows edge detection algorithm. The evaluation performed has shown that a GPU-acceleration improves computing efficiency. Moreover, this has specific applications in areas such as navigation, topographic mapping, and more.

The remainder of this paper is organized as follows. In section 2, we briefly describe related work on edge detection techniques in parallel platforms based on GPU. Section 3 shows preliminaries for edge detection and parallel computing, including a brief overview of CUDA concepts. A parallel implementation of contemplation for our proposed method is presented in section 4. It explores the process of noise detection, restoration, and edge detection with helper operations such as thresholding, max/min operations, convolution, gradient calculations and non-maximum suppression. Additionally, two approaches have been used to implement the convolution process using global and shared memory of the GPU. Experimental results are discussed in section 5. Finally, a closure to our work and discussion of some future trends is presented in section 6.

2. RELATED WORK

Digital image processing is a vital component in medical, industrial, surveillance and commercial applications. The evolution of digital image processing applications demands a high-speed processing power to meet real-time requirements. Consequently, a graphical processing unit accelerator is introduced along with other accelerators [20] in fields such as computer vision, mathematics, encryption [21], and deep learning [22] and has developed year by year to cut off the gap between the throughput and the time consumed. Recently, many research-related investments have been made in graphic design and computer vision to improve the quality of the image by bringing high definition quality in real-time applications [23–24]. Thanks to the parallel computation property that GPU poses, edge detection technique in GPU shows many promising results. Many operations can be parallelized easily and handled by threads in the GPU architecture. Most edge detection techniques can be exploited and parallelized in the GPU platform. Here, some implementations of edge detectors are provided to the reader to show the power of GPU.

Parallel processing algorithms: Sobel edge detection and an homomorphic filter are applied on the image using GPU in [25]. Results are compared with the CPU sequential implementation where GPU outperforms CPU 49 times. They also showed that by increasing the size of the image, the speed is increased. An efficient Sobel detection implemented using GPU is proposed in [26]. The authors conduct the experiment in different platforms such as CPU, GPU, and FPGA. In terms of execution time, the GPU implementation of Sobel detection is outperforming the CPU implementation, which is implemented using C++ language, and the FPGA implementation under Xilinx ISE synthesizer using VHDL.

A performance analysis of Sobel detector has been conducted on a heterogeneous system using OpenCL [27]. The authors implement the Sobel operator in GPU using OpenCL (i.e. open standard framework). Results showed that GPU is highly recommended for parallel computations. In addition, the execution time on GPU outperforms the computation time taken by the CPU. OpenCL has a property that can be implied in any parallel unit regardless of the vendor of that unit.

An implementation of Canny edge detection on GPU using CUDA platform is presented in [28]. The results indicated that the GPU achieve better speedup over the conventional software implementation by sub-dividing the image of size 1024×1024 into smaller images. They used shared memory (caching) rather than a global memory in GPU to reduce the number of reading and writing operations while the input image is loaded into global memory due to small storage of the shared memory. The performance of GPU using CUDA achieved 50 times speed-up of CPU system.

An improved implementation of Canny edge algorithm is proposed by avoiding recursive operations of the conventional Canny edge algorithm in [29]. The experiment has been conducted in various images sizes. The speedup ratio is

increasing by increasing the image size. In addition, the new operator eliminates some of the noise and shoots such that it provides clearer edges.

A GPU-based new parallel approach for accelerating the execution of edge detection is presented in [30]. The existing method for Canny approach is modified to run it in a fully parallel manner. It is done by replacing the breadth-first search procedure with a parallel method. The evaluations showed that the proposed approach on GPU using the CUDA platform improved the speed of execution by 2–100x when compared to CPU-based processing.

A statistical image sampling method based on GPU using edge templates is proposed in [31]. The proposed algorithm attempts to accelerate our up sampling using GPU. This is accomplished by reducing the input resolution-grids dependency artefacts and rebuilding low resolution images to get high-quality up-sampled images in real time. The method is applied for edge detection large scale terrain rendering. However, it is not applied for medical images.

An efficient Canny edge detection using a 2nd derivative in CUDA implementation for the Insight Segmentation and Registration Toolkit (ITK) has been introduced by [32]. The proposed solution was compared with ITK and OpenCV implementations in different Graphics cards (i.e. different GPU generation). Results showed that the proposed solution gives a good result in execution time. Canny implemented using CUDA is marginally better than the Canny version in OpenCV, while CannyCV performs much better than ITK version due to thread parallelization and SSE instructions.

A new level set algorithm has been presented in [33] which is implemented using a GPU that uses edge, region, and 2D histogram information to segment objects in a scene. The native Level Set Method (LSM) is computationally expensive, hence, the proposed algorithm implement Lattice Boltzmann method (LBM), which greatly reduces the computation due to its high nature of parallelization along with the body force to solve the level set equation. It is robust against noise. Results are compared with C-V, Li's and Chen methods. The proposed work is independent to the position of the initial contour and applied in different kind of images, including medical images, such that it shows very good results with the lowest Hausdorff and Martins values. The only limitation mentioned by authors is the memory complexity when storing the distributed functions of Boltzmann method.

As a result of the limitations in classical operators and the imperfection of the traditional Gabor wavelet transform, a parallel version of Gabor wavelet transforms [34] is proposed using CUDA. The proposed solution is compared with Roberts, Canny (TL = 0.3, TH = 0.85), and the serial implementation of the Gabor wavelet transform. The test was conducted using "Lena" 256 × 265 image in Windows platform with one GPU model (9600M GT generation). Results showed that the proposed solution has shortened time.

Generally, solutions with adaptive (combines the first and the second derivative) and accurate edge detection tend to be more computationally expensive than approximation approaches. Parallelization plays an important role in image processing techniques for obtaining high computational power; therefore, a high throughput is gained. Many edge detection stages exploit a degree level of parallelism; therefore, a parallel implementation on GPU is a perfect selection to take the advantage of both the accuracy of the adaptive solution and the speed gained from using the platform.

3. PARALLEL COMPUTING IN GRAPHICAL PROCESSING UNIT (GPU)

Parallel computing is a form of concurrent (simultaneous) execution, where many workloads (tasks) are carried out. In other contexts, this kind of computing is referred to as a computer science discipline that deals with system architectures related to the concurrent execution of applications along the software. Simply, a parallel system is a system which can distribute the workload among individual processors and take care of the computation challenges during processing. The importance of parallelization was indicated back in the late 1950's in the form of supercomputers. There were earlier attempts to start High-Performance Computing (HPC) by using many-core systems to accomplish certain tasks and introduce a shared memory multiprocessor working on a shared data side-by-side. Nowadays, multi-core systems are becoming dominant in the market. The value of parallel computing can be gained through increasing various overall performance metrics such as enhancing speed, optimizing power efficiency and the throughput. Shifting from one core systems with high clock frequencies to multiple core systems with different kind of voltage powers and clock frequencies opens many doors in recent trends.

Currently, vendors are starting to build mobile and high-end platforms that can do many tasks efficiently in terms of speed and power. Today, many systems come with many packed cores (e.g. GPU) that can do the job at extremely high speed [35–37]. Although the field of parallel computing is increasing year after year which agrees somehow with Moore's law in the case of several cores that are packed per chip, certain challenges are raised in these kinds of systems. Increasing and scaling task of the performance of many-core GPUs and CPU means facing obstacles that make the task of improvement is not so naïve Challenges can be discussed and labeled briefly in two main classes as described in [35] and [38].

General Purpose Graphics Processing Unit (GPGPU) is a concept derived from the idea of submitting GPU an ordinary computation that is traditionally done on CPU[34]. It is not a separate type of GPU; instead, it is a software concept. Many GPGPU applications, such as DSP (Digital Signal Processing), DIP (Digital Image Processing), ASP (Audio Signal Processing), bioinformatics, FFT (Fast Fourier Transform) and many others in scientific computing. GPUs have developed into a highly parallel platform with numerous computational powers and high memory bandwidth. Moreover, GPU is now used in many toolboxes provided by certain types of applications using CUDA platforms, such as Adobe Photoshop, 3D Studio MAX, MathWorks, and many other scientific applications. Figure 1 shows the comparison between the CPU and the GPU performance scale in terms of floating-point operation [40].

A GPU is a specialized microprocessor that is dedicated to accelerating 2D-3D operations on rendering graphics [41]. In

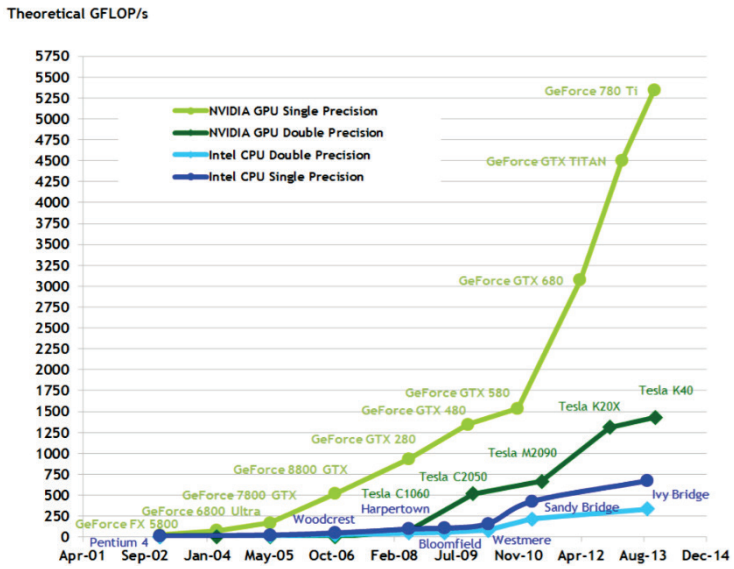

Figure 1 A comparison study by NVIDIA on Floating Operation per second between GPU and CPU.

the beginning, GPUs were devices with a collection of core amounts that had a massive appetite for both computation and bandwidth. They were designed to handle a huge amount of processing in computer vision and graphics; as a result, they are very efficient and cost effective in terms of speed and time on image processing algorithms. GPUs were first promoted by NVIDIA in 1999 with the introduction of GeForce 256. Three years later, ATI was marketing their first Visual Processing Unit (VPU) under codename Radeon 9700. Modern GPUs are very effective and efficient in dealing with image processing. GPUs can be viewed in several forms.

As the development trades on GPU performance increases and the idea of single processor computation is becoming old-fashioned and inadequate, numerous demanding applications such as computer vision, video-game industry, and a wide range of other high-performance applications will be massively dependent on the features that GPUs present. A worthy set of arguments were most of them tackled today by GPU models and introduced by Fayez Gebali [42] could be summarized as follows:

- The performance of computers depends on processing powers while increasing the clocking (aka speed) will consume unacceptable power such that it produces heat and new challenges.
- Developing programming tools that can sense parallelism in given algorithms.
- Memory systems are still slower than processors and their bandwidth is limited.
- The number of processors being deployed as well as the communication overhead.
- Optimization of future computers considering parallel programming at all levels such as algorithms, operating systems, compilers and program developments.

In Desktop PCs, GPUs are presented as PCI-Es video cards. While in mobile devices, such as laptops and modern smartphones, GPUs are embedded in the motherboard. Lately, Intel has started to include an integrated GPU with their same CPU die processors with different codenames like Iris and Pro Graphics [43]. Nowadays, GPUs are introduced into the market with different capabilities and forms. As an example, as it appears in smartphones, GPUs architectures differ from a personal computer (Desktop) or video-game consoles, which are specifically built for low-power management devices working on limited battery power.

One of the forms presented to harness the power of the GPU is an external GPU (eGPU) using specific docks. There is no difference between ordinary GPUs that were sold for desktops and eGPUs in the type of graphics card. Portable machines like laptops demand a massive number of computational units (i.e. ALUs) for advanced and intensive applications. Due to some limitations, such as power management, spacing and cooling down, the existing graphic card is not sufficient, hence, eGPU is usually packed with a separate power supply in chassis to fulfill this need. Although the eGPU solution resolves many issues, the benchmarking tests show that there is a waste of up to 20% of the overall performance due to connectivity and communication between the external GPU and the machine compared to the embedded GPU of the same brand or vendor [44]. In addition, the approach is not cost effective and needs an external power source, an interface connection (thunderbolt, PCI, etc.), and of course a graphics card.

Before concluding this section, it is useful to look at a new trend that was recently developed to overcome the problem of some limited embedded GPUs in mobile machines like a laptop.

Although the proposed solution resolves many issues, the benchmarking tests show that there is a waste of more than 15–20% of the overall performance, due to connectivity and communication issues between the External GPU and the machine, in comparison with the embedded GPU or graphic card based on the same brand or vendor [44]. In addition, the approach is not cost effective and needs an external Power Supply Unit (PSU), an interface connection (thunderbolt, PCI, etc.) and, of course, a graphics card (GPU). After the introductory about the graphics processing unit, it is worth concluding by stating some

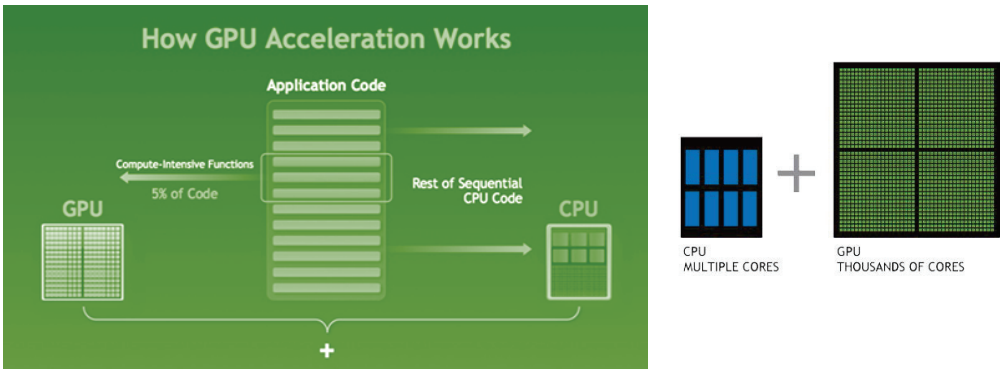

Figure 2 A diagram illustrates the programming model of how the GPU offloading the workload and the variety of cores it has compared to CPU.

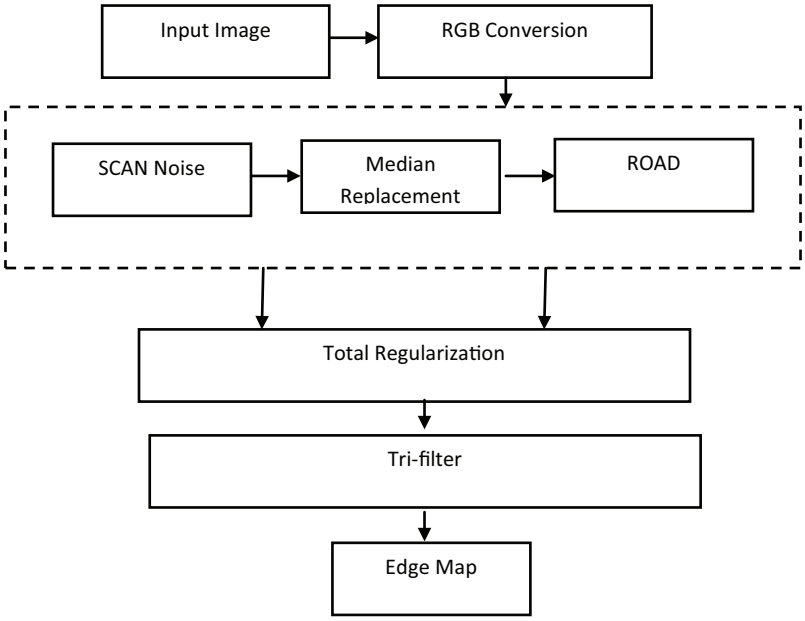

Figure 3 An abstract level design for the proposed operator.

of the benefits. First, using GPUs frees up CPU resources by offloading the workload. Second, a good selection to accelerate embarrassing parallel workloads. Finally, the portability and programmability using a certain type of programming models on GPU.

As shown in Figure 2, refutation of arguments [42] could be illustrated as follows:

1. A GPU is used to free up CPU resources by offloading the workload.
2. A GPU is a best choice to accelerate embarrassing parallel workloads.
3. The portability and programmability by using parallel programming models and development processes.

4. PROPOSED ENHANCED GPU BASED ANTI-NOISE HYBRID EDGE DETECTION METHOD

Digital image processing,such as single thresholding, non-maximum suppression and 2-D convolution for edge detection process has been implemented by exploiting the power of GPUs. We will show a real implementation of image processing algorithms on GPU using various methods of programming models. A break-down structure of code snippets will be given for more illustration without digging into details. Furthermore, a tackle on optimizing a vector and matrix operations is presented by revising loop-based and scalar-oriented code using vectorization that is based on a software approach using CPU and GPU. The next section will show the numerical results between the single and parallel execution of the presented implementations in this section.

The proposed enhanced GPU based anti-noise hybrid edge detection method algorithm is presented in Figure 3, which is derived from [18]. The figure shows the abstract level design of the proposed operator. The input colored image is converted to a grayscale image and filtered using median filters, Ranked-Ordered Absolute Differences (ROAD) statistics and total regularization to remove the noise. The edge is finally detected using operators like Soble and Laplacian.

4.1 Code Syntax and Conventions

In this subsection, many algorithms are presented as pseudo codes in MATLAB and C extension style code and used

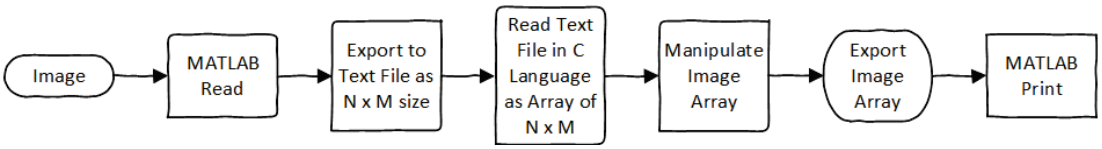

Figure 4 A flow diagram of how the image is read and after applying operations on it using GPU platform.

```
if(i < N && j<N ){
 {if(a[index]<=T)
   c[index]=0;
     else
c[index]=a[index];}

     }
```

(a)

```
[x,y] = size(A);
for i=1:x
    for j=1:y
        if A(i,j)<=T
            A(i,j)=0;
        end
    end
end
image=A;
end
```

(b)

Figure 5 Thresholding implementation using (a) CUDA C. (b) loop-based code style (MATLAB).

interchangeably in the context. The loop-based serial execution is usually represented as MATLAB style code while the parallel implementation is presented in the context as a C extension code (more precisely a CUDA C extension). Parallel implementations are shredded into building blocks (do not confuse with CUDA blocks) and thus, the concentration only deals with the actual logic of the program rather than the initialization issue and definitions. Some operations will be subjectively evaluated as a proof of a consistent result on both implementations. For simplicity, the process of reading or printing images is not taken on consideration significantly during the implementation of CUDA C and MATLAB image processing toolbox API's are used to take care of the reading/showing functions.

As shown in Figure 4, the process of reading image is done using MATLAB toolbox using the imread() function, where the final image is shown using imshow() function after applying set of operations such as convolution, thresholding, and mathematical and logical operation on CUDA C. At this rate, the focus will be on the computational procedures rather than presentations and conversions.

4.2 Embarrassingly Parallel Problems in Image Processing

Embarrassingly parallel problems are sometimes called "pleasingly parallel problems" where little or no effort is needed to separate the task into smaller parallel problems. In this type of problem, there are no dependencies between executed subtasks which build the overall problem. An example of this kind of problem is the task of 3-D projection, where each pixel in the screen is rendered independently.

4.2.1 Single Thresholding

Thresholding is one of the simplest operations that can be run independently on each pixel without any dependency between pixels. Hence, a good speedup will be done using parallel execution where each pixel is compared to a certain threshold value without waiting the previous one is finished as shown in Figure 5(b) on the loop-based version of implementation. In Figure 5(a),each pixel is working alone in 2-D array of an image on the GPU.

As we can see, thresholding implementation is vectorized under the GPU platform using Single Instruction Multiple threads (SIMT) where each thread is holding a pixel and compares it with the threshold value. Furthermore, the logic representation of CUDA C syntax seems to be easier than that shown in Figure 5(b) and C-style code loop-based version. Figure 6 shows two identical outputs of thresholding Lena edge map image using the same implementation on Figure 5(a) and (b) for GPU and CPU, respectively.

4.2.2 Mathematical Operations on Images

Many image processing algorithms composed of a set of mathematical and logical operations such as addition, multiplication, AND-ing, and OR-ing between pixels. As the size of the image increases, these operations consume time and resources. Therefore, the GPU is the best choice to complete this kind of task due to the richness of ALU units that GPU possesses.

4.2.2.1 Image Manipulation and Scalar Product Image manipulation techniques are essential operations in digital image processing. Edge detection is a feature extraction technique and is considered as a high-pass filter that needs, after processing, many enhancements, such as add/remove content to/from the image. Sometimes we need to implement some logical operations such as morphological operations. The result of edge detection is the summation of gradients (2-D arrays). Thus, adding these arrays using a GPU will be in the blink of an eye as shown in Figure A1 in Appendix A.

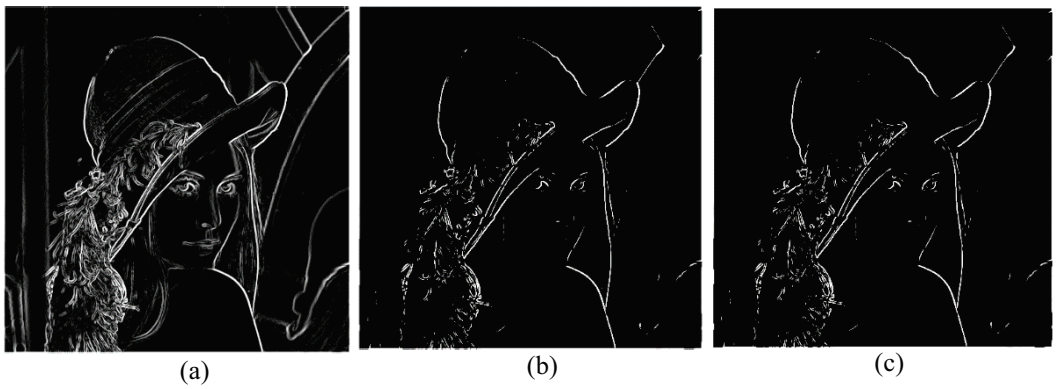

Figure 6 (a) A proposed edge map for Lena. (b) Thresholding edge map (a) for T<180 using the implementation in Figure 4(a). (c) Thresholding procedure is applied on the same image (a) using the loop-based version in MATLAB.

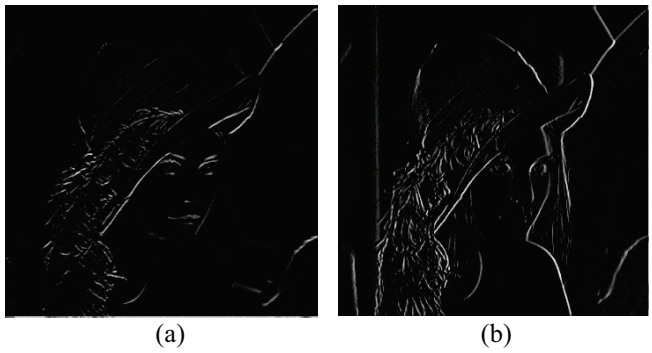

Figure 7 Results of convoluting two operators in −x (a) and −y (b) directions for magnitude calculation.

Figure 8 Gradient image computation using (a) MATLAB (b) CUDA C.

4.2.2.2 Squaring and Square Root The calculation of gradient magnitude for the classical type of operators in edge detection depends proportionally to the square root of the summation of squaring the convoluted result in both –x and –y directions of applying certain operators for the image. The square root function presented in a C library is used to get the value as shown in Figure A2 in Appendix A.

Although the syntax of the implementation using a MATLAB software looks more elegant, the speed gained from GPU implementations is much higher than the speed gained from MATLAB. Moreover, due to the conversion between numerical values, the output from CUDA C is more accurate. Figure 7 shows the two convoluted images Gx and Gy (implementation of results will be shown later) of the enhanced Sobel operator. Figure 8 shows the output images from CUDA and MATLAB execution. Although edge map images shown in Figures 7 and 8 look the same, Figure 7 displays the histogram of both images with more detail between gradient images showing their difference.

While both images look similar in Figure 8, Figure 9 shows not. Essentially, this is a proof of precision used in dealing with numbers in both environments (MATLAB and CUDA C). In CUDA, a precision of a double is used to consider more significant figures rather rounding in MATLAB.

4.2.3 2-D Max and Min Functions

Max function for 2-D arrays such as images is simply the largest/smallest element between a set of 2-D arrays at the same index for all element arrays. Let $A_0, A_1, A_2 \ldots A_n$ are 2-D arrays of size $N \times M$ and let max/min is a 2-D array that can be defined as

$$\max/min\,(A_0, \ldots\ldots, A_n) = \max/min\left(a_{0_{ij}}, \ldots\ldots, a_{n_{ij}}\right);$$
$$0 \leq i, j < N, M$$

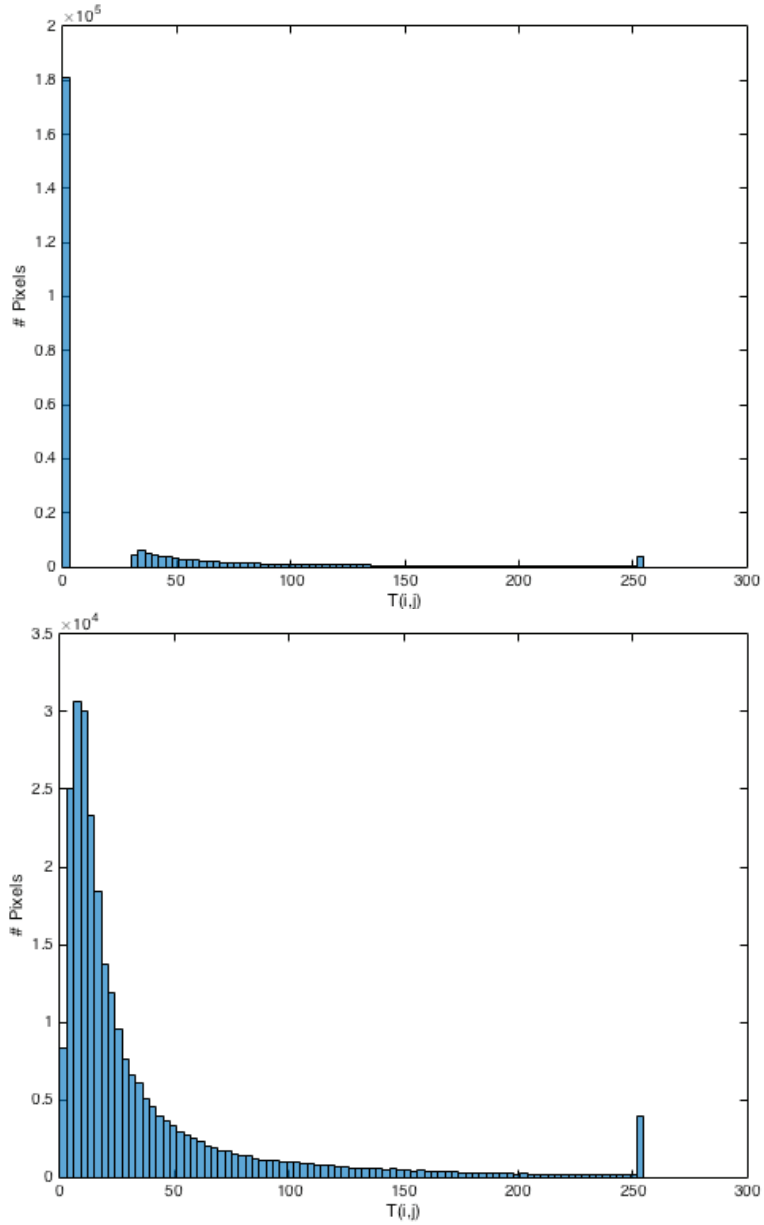

Figure 9 A histogram chart of the magnitude computation gained from (a) MATLAB result (b) CUDA C result.

A good illustration is presented in Figure A3 in Appendix A which describes how the function max is working with two 2-D arrays A and B. The implementation of max function in CUDA C between two images is shown in Figure A4.

Rather than looping around all arrays and checking each index with each iteration and comparing, parallel comparison at the same time for several block of threads is easily done in GPU, where each thread is one comparison between two elements in both arrays.

4.2.4 Non-Maxima Suppression

Non-Maxima Suppression (NMS) is a thinning algorithm based on local processing operations, and is used to reduce the thickness of edge pixels. The idea of NMS is to suppress all gradient values to zero and keeps only local maximal values. In this section, an implementation of NMS is given as a CUDA C program.

Assume both Gx, Gy for gradient calculation are available for further computation such that the returned result of ConvGPU() is a gradient image in direction –x and –y for Gx, Gy, respectively. Thus, the first step is to calculate the inverse trigonometric function (arctan) for both image gradients Ix and Iy. CUDA C APIs have a large set of built-in functions such as math library and so on, inverse tangent is defined as atan2() function. Since we are dealing with float numbers, CUDA C math API thankfully provides a float-type version of arctan which is defined as atan2f() CUDA C function.

The first process is to compute the edge direction for each pixel at location (i,j =index) using atan2f() function.

$$\text{Direction [index]} = \arctan(\text{Y[index], X[index]})x\frac{180}{Pi}\Big)$$
$$if\ Direction\ [index] < 0$$
$$Direction[index] = Direction\ [index] + 360$$

The result returned by atan2f() is defined as radian angles, hence a conversion is needed to get degree angles by multiplying the returned result with (180/PI = 3.14). The last step of

the process is to adjust negative angles by adding 360 if the direction possesses negative value. For each location in gradient images X and Y, the tan inverse is calculated independently using SIMT execution model such that each thread is computing the tan inverse related to its index.

The next process is to update the obtained edge direction array in parallel to be adjusted to nearest angle degree $\in \{0, 45, 90, 135\}$ as shown in the next implementation.

```
if Direction[index] >= 0        and Direction[index] < 22.5
or Direction[index] >= 157.5    and Direction[index] < 202.5
or Direction[index] >= 337.5    and Direction[index] <= 360

                  --> Direction2 [index] = 0

Elseif Direction [index] >= 22.5     and Direction[index] < 67.5
or     Direction [index] >= 202.5    and Direction[index] < 247.5

                  --> Direction2 [index] = 45

Elseif Direction[index] >= 67.5      and Direction[index] < 112.5
or     Direction[index] >= 247.5     and Direction[index] < 292.5

                  --> Direction2[index] = 90

Elseif Direction[index] >= 112.5     and Direction[index] < 157.5
or     Direction[index] >= 292.5     and Direction[index] < 337.5

                  --> Direction2[index] = 135
```

After calculating the adjusted directions, we start the NMS process and compare each magnitude along the obtained directions from previous process. Shown below is how the magnitude is calculated.

```
Magnitude[index] = sqrt(X)^2[index] + Y^2[index]
```

The next step is to consider each adjustment edge direction obtained to get the maximum magnitude by suppressing all other magnitudes along this direction by comparing the magnitude of the current pixel (i,j) with neighborhood pixels.

The result is presented in a binary format. In order to return the original values of each pixel in location (i,j), a dot product is needed between the final result and the acquired magnitude.

```
result[index]=result[index]*Magnitude[index]
```

In the next subsection, a tackle on a 2-D convolution on GPU will be revised to improve the execution speed of the process such as edge detection and operations requiring convolution between signals.

4.2.5 Advanced Problem in Image Processing

The convolution is used without giving any clue or details about the nature of the operations and how they are implemented. This subsection focuses on showing the task of image convolution on the GPU platform. An introduction to the problem is given first graphically and mathematically. Second, a simple way to implement a 2D convolution and finally, an optimized implementation is provided.

4.2.5.1 Image Convolution Convolution is a mathematical operation and a vital component ubiquitous in digital image processing for image restoration, image segmentation, feature extraction, and object pattern recognition. Mathematically, a convolution is an operation on two signals, f and g, that measures the amount of overlap between two signals and can be defined continuously in a time domain as specified in Equation (2):

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2}$$

In a 2-D image function as stated in Equation (3), convolution could be performed discretely using a discrete kernel g such that

$$(f * g)[i, j] = \sum_{n}\sum_{m} f[i-n, j-m]g[n, m] \tag{3}$$

Image convolution is basically a scalar product of the filter weights and all pixels of the image within a kernel as shown in Figure 10.

As shown in Figure 10, the image f is convolved with a kernel of radius 1 (one-ring neighborhood is deliberated throughout the convolution process). In some cases, the kernel is centered at a location (i,j) of the input image where pixels of the original image at location (i+a, j+b) $\{-1 \leq a, b \leq 1\}$ outside the image are treated as zeroes.

4.2.5.2 Convolution on GPU Using CUDA This is the most important part of the work where the convolution procedure is parallelized, enhancing many operations, such as edge detection and image filtering in digital image processing. Initially, a naïve implementation for both CPU and GPU are presented. The naive implementation of 2-D convolution is time exhausted on the CPU. The GPU implementation of 2-D convolution is a lot faster as shown later in the experimental evaluation section.

Later, we enhanced the process of convolution using a shared memory per block. Results of each block are computed locally in the corresponding block and conquered later by writing the value of the new pixel to the global memory. Because of using shared memory approach, the gained speed up is up to a factor of 37.

Naïve convolution on GPU using global memory on CUDA

An implementation of naïve 2-D convolution on a CPU is presented in Algorithm 1. The time complexity of this kind of algorithm are running on $O(n^4)$

GPU is specialized to tackle this type of algorithm by running each block of pixels concurrently, and thus reduce the time complexity of the overall algorithm. Initially, the reduction is greater than $O(n^2)$.

An implementation of naïve 2-D convolution on GPU is presented in Figure A5 in Appendix A which runs on the GPU device using SIMT execution model. As shown in Figure A5, the two outer loops in Algorithm 1 are replaced with an index of the current pixel (i,j) in global memory. Hence, each logical outer loop that is now represented as indices of pixel locations in parallel implementation will run concurrently, and each pixel will execute the inner loops independently.

Convolution on GPU using shared memory on CUDA

Figure 11 shows the relation between the convolution process (top box) with both shared memory and global memory and

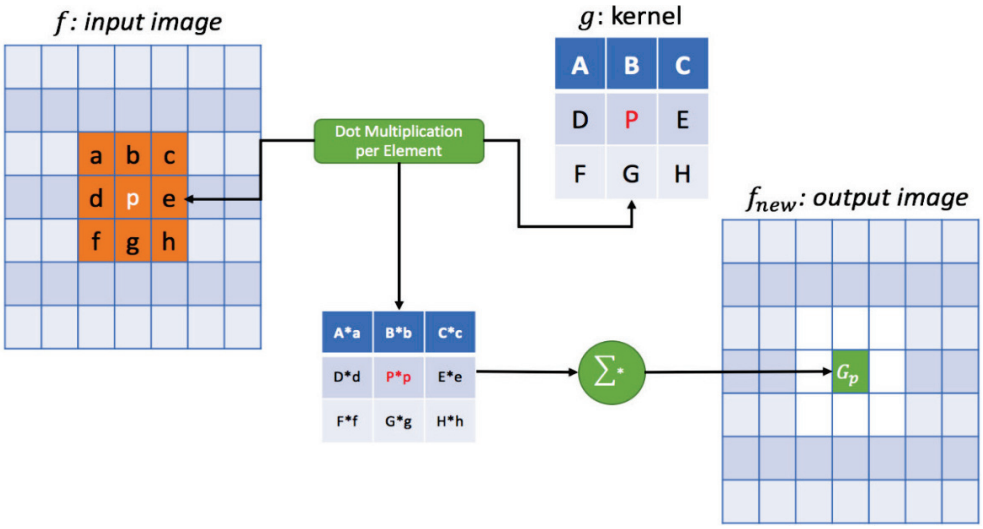

Figure 10 A diagram illustrates the process of a 2-D image f convolved with a 3 × 3 2-D kernel at point p of image f.

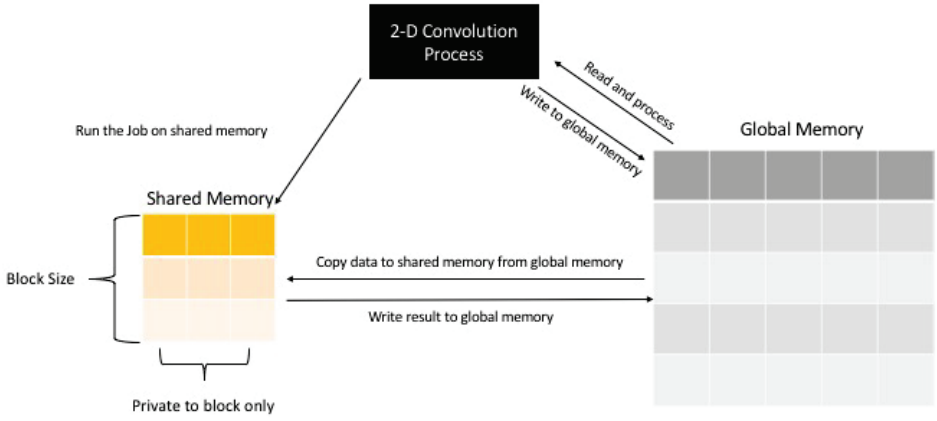

Figure 11 The interaction between three objects: global memory, shared memory and the process (convolution).

Algorithm 1 A serial 2-D convolution on CPU composed of four loops.

```
function 2DConv (Image, kernel)

for y← 0 to y < Image height do
   for x← 0 to x < Image width do
         sum ← 0;
     for j← 0 to j < kernel height do
      for i← 0 to I < kernel width do
       sum ← sum + Image(y+j,x+i)*kernel(j,i)

      Image(y,x) ← sum

end function
```

how data flows. Convolution process is the kernel core where each executed block holds the shared memory. Each block will write to the global memory after working locally on the shared memory data. Once the result is generated inside the block, the kernel will write back the result to the global memory, such that it reduces the amount of communication between kernel and global memory. Initially, each block will warp the workload (pixels) from the global memory. Once all blocks have gathered the data, each block will run independently and write down the result to the original location in global memory after finishing execution as shown in Figure A6 in Appendix A.

Convolution is a serious operation in digital signal processing. For a large set of signals (e.g. images), time becomes a critical issue and optimization is necessary. Generally, a lot of enhancement in image convolution under GPU has been studied. However, not all operators are capable of being applied practically due to some characteristics they possess (e.g. separable filtering).

One sort of improvement for our implementation is to enhance the execution of convolution operation under a graphical processing unit using shared memory. Fortunately, CUDA platform makes the programmer able to easily use the shared memory within each block without involving any assembly and low programming APIs. The shared memory's own set of characteristics which make the implementer control threads per block.

So far, an illustration of a 2-D convolution implementation is studied and tackled on GPU using global memory (native) and shared memory approaches. A Total Variation (TV) denoising problem is a classic example of prior analysis that grasps for all linear transforms, and can be expressed for the sake of convenience using Equation (4):

$$\arg\min_{u} |u_0 - u|_q^q + \lambda |D_V(u)|_q^p + \lambda |D_H(u)|_q^p \quad (4)$$

Clearly, a TV problem is a finite difference transform where D_V and D_H are vertical and horizontal finite difference operators. Similarly, λ is the same regularization parameter that appears in Equation (4), and p and q are constants span from 0 to 1. Since the impulse noise is sparse, the value of q is on range (0,1]. Here, we are proposing the eROAD algorithm that has been shown above with a split-Bregman algorithm and soft-thresholding to solve Equation (4).

The first step needed is to substitute some terms such that the problem in Equation (4) becomes a constraint optimization problem. Let $z = u_0 - u$, while $v = D_V(u)$, and finally $w = D_h(u)$. The problem then can be rewritten as following:

$$\min_{u,z,v,w} |z|_q^q + \lambda |v|_p^p + \lambda |w|_p^p \quad (5)$$

$$subject\ to:$$
$$z = u_0 - u$$
$$v = D_V(u)$$
$$w = D_h(u)$$

Weak penalty function can be introduced to the problem in Equation (5) to become an unconstraint problem and the formula could be re-expressed as the following

$$\min_{u,z,v,w} |z|_q^q + \lambda |v|_p^p + \lambda |w|_p^p + \underbrace{\mu_1 |z - u_0 + u|_2^2 + \mu_2 |v - D_V(u)|_2^2 + \mu_2 |w - D_h(u)|_2^2}_{weak\ penalty\ function} \quad (6)$$

By applying a split-Bregman on Equation (6), three variables are introduced to control the update value as following:

$$\min_{u,z,v,w} |z|_q^q + \lambda |v|_p^p + \lambda |w|_p^p + \mu_1 |z - u_0 + u - a^i|_2^2 + \mu_2 |v - D_V(u) - b^i|_2^2 + \mu_2 |w - D_h(u) - c^i|_2^2$$

where **a, b, c** are updated as following:

$$a^{i+1} = a^i + u_0 - z - u$$
$$b^{i+1} = b^i + D_V(u) - v$$
$$c^{i+1} = c^i + D_h(u) - w$$

Hence, the problem can be solved for (u, z, v, w) independently such that each variable considered as a problem itself as stated in Equations (7–11).

$$\arg\min_{z} |z|_q^q + \mu_1 |z - u_0 + u - a^i|_2^2 \quad (7)$$

$$\arg\min_{v} \lambda |v|_p^p + \mu_2 |v - D_V(u) - b^i|_2^2 \quad (8)$$

$$\arg\min_{w} \lambda |w|_p^p + \mu_2 |w - D_h(u) - c^i|_2^2 \quad (9)$$

$$\arg\min_{u} \mu_1 |z - u_0 + u - a^i|_2^2 + \mu_2 |v - D_V(u) - b^i|_2^2 + \mu_2 |w - D_h(u) - c^i|_2^2 \quad (10)$$

$$\arg\min_{u} |I - u|_2^2 + \lambda |u|_p^p \quad (11)$$

where I denotes the original signal and u is the estimated signal.

In case of Equation (11), the equation can be differentiated as a differentiable convex optimization w.r.t x such that

$$u(\mu_1 A + \mu_2 D_V^T D_V + \mu_2 D_h^T D_h) = \mu_1 (u_0 - z + a^i) + \mu_2 (D_V^T (v - b^i) + D_h^T (w - c^i)) \quad (12)$$

As we can see, the estimated signal u that annotates the reconstructed image appears in Equation (12) which gives an indication of direct step involvement.

5. EVALUATION AND DISCUSSIONS

In this section, we demonstrate parallel processing by means of a graphics processing unit versus sequential processing using central processing unit. The evaluation is steered on a different operation that is used in edge detection, such as image convolution thresholding and other related processes. Results show that the GPU is superior in terms of low execution time and extreme speedup. Moreover, an optimized MATLAB technique called "vectorization" is used against the loop-based and GPU Single Instruction Multiple Thread (SIMT) where the latter outperforms all in terms of speed and the simplicity of code

5.1 Experimental Setup and Machine Configuration

Experiments are done using the scientific software MATLAB that is used for technical computing and simulation. The ease of writing code and the massive libraries available in the software for various kinds of computations are reasons for using this software as a proof of concept of the work. MATLAB provides libraries of parallel functions where they are using GPU integrated with CUDA platform to accomplish a task. CUDA C is provides us the control for memory allocation and selection of blocks-threads combination, while MATLAB does not. All experiments are tested under machine configuration for CPU and GPU as shown in Table 1 and Table 2.

5.2 Image Dataset

Experimental results and evaluation are gained by applying different sets of algorithms including the proposed algorithm on image restoration and edge detection on a set of subjects. There are two categories of image subjects used in experimental results; the first category is the general type of image, such as Lena and Cameraman, obtained from the Gonzalez image database [40]. Peppers and fishing boat are obtained from the USC-SIPI image database [41]. Figure 12 shows the first category of image dataset.

5.3 Time Execution for Edge Detection

Time execution is a definite issue in many applications including digital image processing. In this section, a study on edge detection on CPU and GPU is done to show the power of GPUs

Table 1 Machine configuration for the CPU used for serial execution.

Processor Type	CPU
Generation / Speed	Intel core i7 / 2.3 GHz
Memory	DDR3 / GDDR5

Table 2 Machine configuration for the GPU used for parallel implementation.

Processor Type	GPU
Generation	NVIDIA GeForce GT 750M
MaxThreadsPerBlock	1024
Compute Capability	3.0
MaxSharedMemPerBlock	49152
Memory	DDR3 16 GB / 1600MHz
Driver Version	7.5
Toolkit Version	6.5

(a) (b) (c) (d)

Figure 12 Images dataset used for evaluation (a) Lena, (b) Peppers, (c) Cameraman, (d) Fishing boat.

over CPUs for well-known computational problems such as the convolution process and many similar scientific problems. The performance evaluation is tested under Nsight IDE profile for time execution analysis. An alternative way to compute the time execution for a certain kernel in CUDA platform is to use events in CUDA RUNTIME APIs as shown in Figure 13.

The sequential execution of edge detection implementation is done using MATLAB software. In contrast, a Hyper-Threading (HT) implementation is conducted in the work using the latter software. HT offers optimized methodology through vectorization and usage of all available cores in the machine (other than GPU cores) to execute the program.

The execution time is computed in MATLAB using tic and toc built-in functions. Results are compared with the profiler tool for consistency and reliability where both methods show almost the exact result. Table 3 shows a study on convolution process and how organizing threads per blocks affect the performance of execution in two different approaches.

```
cudaEvent_t beginEvent;
cudaEvent_t endEvent;
cudaEventCreate(andbeginEvent);
cudaEventCreate(andendEvent);
cudaEventRecord(beginEvent, 0);
// Computational Kernel
cudaEventRecord(endEvent, 0);
```

Figure 13 An alternative solution to compute the execution time for a CUDA kernel.

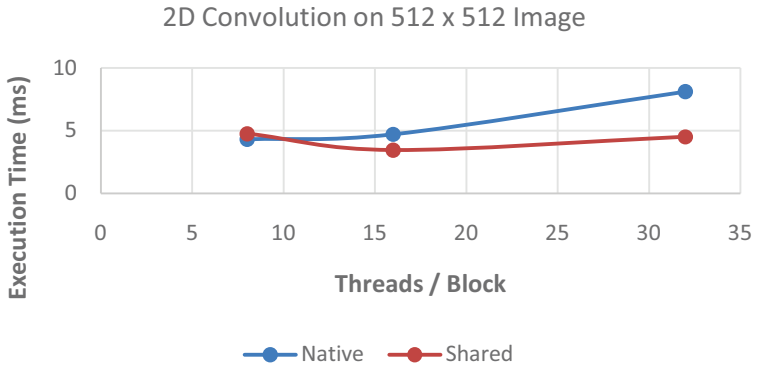

Figure 14 Execution time for shared and native memory applied on 512^2. Image using GPU.

Table 3 Different image sizes are tested on various block sizes in kernel configuration on a 2-D grid.

Lena Sample	Time execution (ms)		
Convolution 3 x 3	GPU		
Image Size	Threads/Block	Native (Global Mem)	Shared Memory
128 × 128	8	0.285	0.310
	16	0.344	0.241
	32	0.531	0.108
256 × 256	8	1.24	1.18
	16	1.13	0.882
	32	2.056	1.103
512 × 512	8	4.30	4.75
	16	4.7	3.44
	32	8.104	4.5

As observed in Table 3, increasing the number of threads per blocks is not always cooperative. Clearly, GPU implementation using shared memory approach outperforms the native approach by nearly 45–50% speedup latency on increasing threads per blocks. Speedup can be obtained as introduced in Equation (13)

$$Speedup_{latency} = \frac{ExecutionTime_{old}}{ExectutionTime_{nnew}} \qquad (13)$$

Time execution (ms) for various threads for GPU native (global memory) and shared memory is given in Figure 13. It shows that the shared memory takes less execution time.

In Table 4, an edge detection process is evaluated using four approaches on CPU and GPU. In the GPU implementation, we compare the best configuration obtained from Table 4 in both approaches, global memory and shared memory, against native (single core) and HT implementation. Results show that the fastest execution time is primary for the shared memory and the native GPU implementation is secondary. The speedup of edge detection on GPU over naïve implementation on CPU is 37 times using the shared memory approach and 1.5 times over HT implementation.

6. CONCLUSIONS AND FUTURE WORK

Edge detection is a widely-used technique in digital image processing for various scientific applications to detect features that characterize an image. Image processing techniques are pleasingly parallel problems that can be executed in parallel as shown in section 4 algorithms. Thus, parallel implementations are used to harness the power of a GPU platform as a general-purpose computing platform. In our proposed method, our evaluation execution is done after unrolling sequential loops on the parallel platform against the sequential implementation. We evaluated our proposed solution under two approaches of GPU implementation, the naïve approach, where data is processed directly in global memory, and shared memory, where data is processed locally per block. The proposed edge detection method under GPU using global memory direct access is up to 25 times faster than the native CPU implementation, while in using the shared memory approach, the speed gained is up to 37 times faster, outperforming all earlier implementations. As a part of future work, we plan to use larger dataset of images and perform further evaluation.

Table 4 Edge detection time execution is shown against different approaches and software implementations.

Edge Detection Phase	Time execution			
	GPU		CPU	
Image Size	Native (Global Mem)	Shared Memory	Native	MATLAB (Hyperthreading)
128 × 128	1.4	0.992	94.348	2.2
256 × 256	5.53	3.42	333.441	6.35
512 × 512	22.26	15.059	558.67	25.234

REFERENCES

1. Bao, P., Zhang, L., and Wu, X. (2005). Canny edge detection enhancement by scale multiplication. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **27**(9): 1485–1490.
2. Basu, M. (2002). Gaussian-based edge-detection methods-a survey. *IEEE Transactions on Systems, Man and Cybernetics*, Part C (Applications and Reviews). 32(3): 252–260.
3. Xu, Q., Varadarajan, S., Chakrabarti, C., and Karam, L. J. (2014). A Distributed Canny Edge Detector: Algorithm and FPGA Implementation. *IEEE Transactions on Image Processing*, **23**(7): 2944–2960.
4. Canny, J. (1986). A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. PAMI-**8**(6): 679–698.
5. Roberts, L. G. (1980). Machine perception of three-dimensional solids. New York: Garland Pub.
6. Prewitt, J.M.S. (1970). Object Enhancement and Extraction. Picture processing and Psychopictorics. Academic Press.
7. Chidiac, H., and Ziou, D. (1999). Classification of Image Edges. Vision Interface'99. Canada: Troise-Rivieres, 17–24.
8. Jayaraman, S., Esakkirajan, S., and Veerakumar, T. (2009). Digital image processing. New Delhi: Tata McGraw Hill Education.
9. Marr, D., and Hildreth, E. (1980). *Theory of Edge Detection*. Proceedings of the Royal Society of London. Series B, Biological Sciences. 207(1167): 187–217.
10. Wang, X. (2007). Laplacian Operator-Based Edge Detectors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **29**(5): 886–890.
11. Fu, W., Johnston, M., and Zhang, M. (2014). Low-Level Feature Extraction for Edge Detection Using Genetic Programming. *IEEE Transactions on Cybernetics*, **44**(8): 1459–1472.
12. Kai-Jian, X., Yu-Feng, Y., Jin-Yi, C., and Shan, Z. (2010). *An edge detection improved algorithm based on morphology and wavelet transform*. 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE). 404–407.
13. Kimm, H., Abolhassani, N., and Lee, F. (2013). *Edge Detection and Linking Pattern Analysis Using Markov Chains*. 2013 IEEE 16th International Conference on Computational Science and Engineering. 1146–1152.
14. Melin, P., Gonzalez, C. I., Castro, J. R., Mendoza, O., and Castillo, O. (2014). Edge-Detection Method for Image Processing Based on Generalized Type-2 Fuzzy Logic. *IEEE Transactions on Fuzzy Systems*, **22**(6): 1515–1525.
15. Yingjie, Z., and Liling, G. (2007). *A Simple and Efficient Multiscale Edge Detection Method Based on Image Diffusion*. Workshop on Intelligent Information Technology Application (IITA 2007), 235–238.
16. Zhang, Y., and Han, Q. (2011). *Edge Detection Algorithm Based on Wavelet Transform and Mathematical Morphology*. 2011 International Conference on Control, Automation and Systems Engineering.
17. Biemond, J., and Gerbrands, J. J. (1979). An Edge-Preserving Recursive Noise-Smoothing Algorithm for Image Data. *IEEE Transactions on Systems, Man, and Cybernetics*, **9**(10): 622–627.
18. Sa'ed Abed, Mohammed H Ali and Mohammad Al-Shayeji (2017), An adaptive edge detection operator for noisy images based on a total variation approach restoration, *Computer Systems Science & Engineering*, **32**(1): 21–33.
19. Cheng, Jianghua, Wenxia Ding, Xiangwei Zhu, and Gui Gao. *GPU-accelerated main road extraction in Polarimetric SAR images based on MRF*. In Industrial Electronics Society, IECON 2016–42nd Annual Conference of the IEEE, pp. 928–932. IEEE, 2016.
20. Haji Rassouliha, A., Taberner J.,Nash, M. and Nielsen, P. (2018). Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms. *In Signal Processing: Image Communication*. **68**(2018):101–119.
21. Haji Hassani, O., Monfared, S., Khasteh, S. and Gorgin, S. (2019). Fast AES Implementation: A High-Throughput Bitsliced Approach. *In IEEE Transactions on Parallel and Distributed Systems*. **30**(10): 2211–2222.
22. Mittal, S. and Vaishay, S. (2019). A survey of techniques for optimizing deep learning on GPUs. *In Journal of Systems Architecture - Science Direct (99)*.
23. Esquembri, S., Nieto, J., Ruiz, M., De Garcia, A. and De Arcas, G. (2018). Methodology for the implementation of real-time image processing systems using FPGAs and GPUs and their integration in EPICS using Nominal Device Support. In *Fusion Engineering and Design*. **130**(2018): 26–31.
24. Blug, A., Regina, D., Eckmann, S., Senn, M., Bertz, A., Carl, D., and Eberl, C. (2019). Real-Time GPU-Based Digital Image Correlation Sensor for Marker-Free Strain-Controlled Fatigue Testing. *In applied Science 2019*. **9**(2025): 1–15.
25. Zhang, N., Chen, Y., and Wang, J. (2010). *Image parallel processing based on GPU*. 2010 2nd International Conference on Advanced Computer Control, 367–370.
26. Chouchene, M., Sayadi, F. E., Said, Y., Atri, M., and Tourki, R. (2014). Efficient implementation of Sobel edge detection algorithm on CPU, GPU and FPGA. *International Journal of Advanced Media and Communication IJAMC*, **5**(2/3): 105–117.
27. Dore, A., and Lasrado, S. (2014). Performance Analysis of Sobel Edge Filter On Heterogeneous System Using Opencl. *International Journal of Research in Engineering and Technology IJRET*, **3**(15): 53–57.
28. Ogawa, K., Ito, Y., and Nakano, K. (2010). *Efficient Canny Edge Detection Using a GPU*. 2010 First International Conference on Networking and Computing. 279–280.
29. Niu, S., Yang, J., Wang, S., and Chen, G. (2011). *Improvement and parallel implementation of canny edge detection algorithm based on GPU*. 2011 9th IEEE International Conference on ASIC. 641–644.
30. Emrani, Zahra, Soroosh Bateni, and Hossein Rabbani (2017). A New Parallel Approach for Accelerating the GPU-Based Execution of Edge Detection Algorithms. *Journal of medical signals and sensors*, **7**(1)18: 33.
31. Zheng, Xin, Chenlei Lv, Qingqing Xu, Peipei Pan, and Ping Guo (2017). A GPU-based statistical image up-sampling method by using edge templates. *International Journal of Computational Science and Engineering*, **14**(1): 64–73.

32. Lourenco, L. H., Weingaertner, D., and Todt, E. (2012). *Efficient Implementation of Canny Edge Detection Filter for ITK Using CUDA*. 2012 13th Symposium on Computer Systems. 33–40.
33. Balla-Arabe, S., Gao, X., and Wang, B. (2013). GPU Accelerated Edge-Region Based Level Set Evolution Constrained by 2D Gray-Scale Histogram. *IEEE Transactions on Image Processing*, **22**(7): 2688–2698.
34. Wu, Q., Fu, Z., Tong, C., and Wang, Q. (2010). *The method of parallel Gabor wavelet transform edge detection based on CUDA*. 2010 The 2nd Conference on Environmental Science and Information Application Technology, 537–540.
35. Almasi, G. S., and Gottlieb, A. (1989). Highly parallel computing. Redwood City, CA: Benjamin/Cummings.
36. Microsoft. (2007). The Manycore Shift. Microsoft Parallel Computing Ushers Computing into the Next Era. 2–7.
37. Unnikrishnan, P., Barton, K., and Chen, T. (2012). *7th Workshop on Challenges for Parallel Computing*. In Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '12), ACM, 251–252.
38. Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. (2011). GPUs and the Future of Parallel Computing. *IEEE Micro.*, **31**(5): 7–17.
39. Fung, J., and Mann, S. (2004). *Computer vision signal processing on graphics processing units*. IEEE International Conference on Acoustics, Speech, and Signal Processing.
40. NVIDIA. (2016). Programming Guide. CUDA Toolkit Documentation. Retrieved February 12, 2016, from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
41. Tse, J. (2012). Image Processing with CUDA. Las Vegas: University of Nevada. 3–26.
42. Gebali, F. (2011). Algorithms and Parallel Computing. Published by Jhon Wiley & Sons, Inc., Hoboken.
43. Intel. (2006). Groundbreaking hybrid architecture for increased performance and flexibility for delivering a compelling graphics and video experience. Intel's Next Generation Integrated Graphics Architecture –Intel® Graphics Media Accelerator X3000 and 3000. 2–14.
44. Techradar. (2016). How to make an external laptop graphics adaptor. Retrieved February 12, 2016, from http://www.techradar.com/news/computing-components/graphics-cards/how-to-make-an-external-laptop-graphics-adaptor-915616.
45. Gonzalez, R. C., and Woods, R. E. (2002). Digital image processing. 2nd edition. Prentice Hall.
46. USC - Viterbi School of Engineering - Site Home. (2016). Retrieved February 12, 2016, from http://sipi.usc.edu/.

APPENDIX A

```
if(i < N && j < N )
{
d[index]=
K*a[index]+b[index]+c[index];
}
```

(a)

```
for i=1:a
for j=1:b
D(i,j)=K*A(i,j)+B(i,j)+C(i,j);
end
end
```

(b)

```
D=K*A+B+C;
```

(c)

Figure A1 Three different implementations of add operation of three 2-D arrays of images using (a) CUDA, (b) loop-based version, (c) MATLAB vectorization.

$$A = \begin{pmatrix} 20 & 2 \\ 130 & -3 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & -2 \\ 255 & -1 \end{pmatrix}$$

$$\max(A,B) = \begin{pmatrix} 20 > 2 & 2 > -3 \\ 255 > 133 & -1 > -3 \end{pmatrix} = \begin{pmatrix} 20 & 2 \\ 255 & -1 \end{pmatrix}$$

Figure A2 The maximum of two 2-D arrays is an array where individual maximum per element array is calculated at the same location for both arrays.

```
if(i < N && j < N )
{
if (a[index]>=b[index])
c[index]=a[index];
else
c[index]=b[index];
}
```

Figure A3 Max function implemented using CUDA C.

```
if(i < N && j < N ){

c[index] =
sqrt(a[index]*a[index]+b[index]*b[index]);}
```

(a)

```
R1 = (GhI.^2);
R2 = (GvI.^2);
Gs = (R1+R2)^0.5;
```

(b)

Figure A4 Two different implementations of finding the magnitude operation of two 2-D arrays of images using (a) CUDA, (b) MATLAB implementation.

```
int i = blockIdx.x*blockDim.x+threadIdx.x;
int j = blockIdx.y*blockDim.y+threadIdx.y;
if (j < 2 || i <2 || j>= h-3 || i>=w-3)
return;
int G[3][3] = {-1,0,1,-2,0,2,-1,0,1};
float Sum;

Sum=0;
for (int a=-1;a<=1;a++)
{
  for (int b=-1;b<=1;b++){
     Sum=Sum+image[(j+b)*w+(a+i)]*G[a+1][b+1];
  }
 }
result[j*w+i]=Sum;
```

Figure A5 An implementation of naïve convolution on GPU by accessing global memory directly.

```
        __shared__ float data[TPB+3][TPB+3];
       int idx = blockIdx.x*blockDim.x+threadIdx.x;
       int idy = blockIdx.y*blockDim.y+threadIdx.y;
       float sum=0;
       float Sum=0;
if (idx<N && idy<N)
{
       data[threadIdx.x][threadIdx.y]=img[idx*w+idy];
       if(threadIdx.y > (N-k))
              data[threadIdx.x][threadIdx.y+k]=img[idx*w+(idy+k)];
       if(threadIdx.x > (N-k))
                     data[threadIdx.x+k][threadIdx.y]=img[(idx+k)*w+(idy)];
       if(threadIdx.x > (N-k) && threadIdx.y >(N-k))
                     data[threadIdx.x+k][threadIdx.y+k]=img[(idx+k)*w+(idy+k)];
       syncthreads();
for (int i=0;i<k;i++){for (int j=0;j<k;j++)
{Sum = Sum+data[threadIdx.x+i][threadIdx.y+j]*Gx[i][j];}
}
       result[idx*w+idy]=sum;
}
```

Figure A6 A convolution process with a shared memory as a processing data set rather than using global memory directly.