PARALLELIZING EDGE DRAWING ALGORITHM on CUDA

Ozgur Ozsen   Cihan Topal   Cuneyt Akinlar

Anadolu University, Computer Engineering Department
{oozsen, cihant, cakinlar}@anadolu.edu.tr

ABSTRACT

Parallel computing methods are very useful in speeding up algorithms that can be divided into independent subtasks. Traditional multi-processor architectures have limited use due to their high cost and difficulties of their use. Recently, Graphics Processor Units (GPUs) has opened up a new era for general purpose parallel computation. Among many GPU programming frameworks, Compute Unified Device Architecture (CUDA) seems to be the most widely used GPU architecture due to its low cost and ease of use. In this paper, we show how to implement our recently proposed novel edge segment detector, the Edge Drawing (ED) algorithm, in CUDA, and present performance studies demonstrating the performance gains in the CUDA architecture compared to a uniprocessor CPU implementation. The results show that a CUDA implementation improves the running time of ED by up to 12x and ED runs at an amazing blazing speed of about 1ms on a 512x512 image. ED is run on different CUDA cards and the performance results are presented.

***Index Terms*—** Parallel image processing, edge detection, real time, GPU programming, CUDA

1. INTRODUCTION

Edge detection is a basic yet very fundamental problem in image processing. Many image processing and computer vision applications start by detecting the edges of a given image as an initial pre-processing step. The detected edges are then combined into edge segment form by a connected component analysis and the produced edge segments are used in such applications as shape recognition, segmentation, tracking and many others [1–7].

A good edge detector must not only produce high-quality edge maps, but it must also run very fast. The speed is especially important in real-time applications. Recently, we proposed a novel proactive edge segment detector that runs in real-time and produces high-quality edge segments [8]. Unlike traditional edge detectors, our edge detector first spots a set of gradient extrema points (called the anchors), and literally draws edges between successive anchors by a heuristic smart routing algorithm, hence the name Edge Drawing (ED) [8]. ED runs in real-time for many image types and sizes in a traditional uni-processor architecture, and produces very high quality edge segments. ED has been shown to run faster than the fastest known edge detector; namely, the OpenCV implementation of the famous Canny edge detector [8].

Although general-purpose GPUs have been in usage for a few years, they have been widely used in real-time computation, and a lot of applications have been ported to GPU-based frameworks for parallel computation. Recently, Canny edge detection algorithm have been ported to CUDA with the structure of a multi-step algorithm [9]. Authors compare their CUDA implementation of Canny to OpenCV implementation of Canny and obtain good speedups. In another study [10], FPGA, CPU and GPU performance have been compared and the GPU have been shown to give better performance compared to others. CUDA implementation of the ISO standard video formats, i.e., Motion JPEG2000, has been shown to perform 20.7 times faster than a strictly sequential CPU implementation [11]. In motion estimation and digital video encoding/decoding applications, the detection of motion vectors is known to be a very important stage. In [12], full-search algorithms and the diamond search algorithms are implemented on CUDA platforms and 8 times speedup have been obtained compared to a CPU-based implementation.

In this article, we show how our recently proposed edge segment detection algorithm, the Edge Drawing (ED) [8], can be implemented in the CUDA framework and a very high-speed edge detector can be obtained. The rest of the paper is organized as follows: In section 2, an overview of ED is presented and the details of ED's implementation on CUDA are given. Section 3 is the experimentation section and presents the running time performance of ED on various CUDA cards.

2. EDGE DRAWING (ED) on CUDA

2.1. Edge Drawing (ED) Overview

Edge Drawing (ED) runs in four major steps [8]:

- Image Smoothing
- Determination of Edge Areas and Edge Directions
- Computation of Edge Anchor Points
- Linking of Anchor Points with smart routing

2.2. Edge Drawing (ED) Steps on CUDA

To implement ED on CUDA, we collected all steps within a few CUDA kernels rather than implementing many kernels for separate steps. This allowed us to keep the local results in shared memory and reduce the number of accesses to the slow global memory, which resulted in big speed gains. It is possible to have 512 threads within each kernel block. But to get square image blocks, at most 16x16 = 256 threads are used. Thus the original image is partitioned into 16x16=256 pixel blocks, and each thread in a block is responsible for the computation of a single pixel in the image.

In the first version of the algorithm, we have implemented each step of ED as a separate CUDA kernel. We then observed that this causes many accesses to global memory during termination of a previous CUDA kernel and invocation of the next. To remedy this problem, we have tried different alternatives and realized that the best results are obtained with three CUDA kernels. That is why the entire ED algorithm is implemented in three CUDA kernels.

2.2.1. Image Smoothing in CUDA Kernel

Like most edge detection algorithms, ED starts with applying a low pass filter to suppress noise. In this step, the standard 5x5 Gaussian filter with σ = 1 is applied. To perform 5x5 Gaussian smoothing on a 16x16 image block, we need to have a 20x20 image block. We need two rows and two columns padding from each direction. These pixels are called the apron pixels in the CUDA terminology. Since 16x16 pixel block is maintained in the shared memory of the Streaming Multiprocessor (SM) for speed purposes, the apron pixels are loaded up from the global memory before the computation starts.

At the beginning of the computation, each thread loads its pixel and one apron pixel from the global memory. All threads are then synchronized before the smoothing computation begins. The computation is quite easy. Each thread in the 16x16 block computes the smoothed pixel value for the pixel it is responsible for, where each thread can work independently of the others.

2.2.2. Determination of Edge Areas and Edge Directions in CUDA Kernel

The next step of ED is to compute the gradient magnitude and edge direction maps. The main purpose of this step is to find areas of potential edges. We use the Sobel operator and compute the gradient values Gx and Gy at each pixel. Since Sobel uses a 3x3 mask for gradient computation, one row/column padding is required in this step. The padding pixels must come from the neighboring blocks, whose Gauss filtering is independent from the current block. Therefore, we have to make sure that all thread blocks performing the Gaussian filtering must finish before we start performing the gradient computation.

To make this possible, we have implemented the Gaussian filtering as one CUDA kernel. When this kernel finishes, we start a new CUDA kernel to perform the gradient map computation. The reason for using two different CUDA kernels is because it is not possible to synchronize threads belonging to different CUDA kernels. Had we performed Gaussian filtering and gradient map computation within the same CUDA kernel, those threads working on the 16x16 block boundaries might have gotten incorrect results from the neighboring blocks, so their gradient computation would have produced incorrect results. This is why we use two separate CUDA kernels: After the CUDA kernel performing the Gaussian filtering finishes up, we start the CUDA kernel that would perform the gradient map computation. Thus all pixel values would be correct in the gradient computation step.

Since the Sobel operator uses a 3x3 filter, only one row/column padding in each direction is enough for the gradient magnitude and direction computation. So each thread within a CUDA kernel block loads up its pixel and one neighboring pixel from the global memory and performs gradient computation. The computed gradient map is then thresholded by a user-supplied threshold. The goal is to eliminate weak pixels where an edge element (edgel) may not be located. The remaining pixels are called the "edge areas" of the image. The final edge map produced by ED will be a subset of pixels from these "edge areas" pixels. Together with the gradient magnitude, the edge direction is also computed as follows: If |Gx| >= |Gy|, a vertical edge is assumed to pass through the pixel. Otherwise, a horizontal edge is assumed to pass through the pixel. So, the gradient angle at a pixel is assumed to be either 0 or 90 degrees [8]. The gradient magnitude and direction maps will be used in the next step; that is, the computation of the anchors. So, they are kept in the shared memory for fast access.

2.2.3. Computation of Edge Anchor Points in CUDA Kernel

After the computation of the edge areas image, ED follows a very unorthodox approach: Instead of testing individual pixels within the edge areas for being edgels, ED first spots a subset of pixels (called the anchors) and then links these anchors by a heuristic smart routing algorithm. Intuitively, anchors correspond to peaks (maximas) of the gradient map [8].

TABLE I. ALGORITHM to TEST for AN ANCHOR

```
Symbols used in the algorithm:
row: threadIdx.y; col : threadIdx.x; [row,col]: Thread Index;
G: Gradient map; D: Direction map;

IsAnchor(row, col, G, D, ANCHOR_THRESH){
    if (D[row, col] == HORIZONTAL){ // Compare with up & down
            if (G[row, col] – G[row-1, col] >= ANCHOR_THRESH &&
           G[row, col] – G[row+1, col] >= ANCHOR_THRESH) return true;

    } else {   // VERTICAL EDGE. Compare with left & right.
            if (G[row, col] – G[row, col-1] >= ANCHOR_THRESH &&
           G[row, col] – G[row, col+1] >= ANCHOR_THRESH) return true;
    } //end-else

    return false;   // Not an anchor
}  //end-IsAnchor
```

Table I shows ED's currently employed anchor detection algorithm. The idea is to simply compare a pixel's gradient value with two of its neighbors in the gradient direction and take a pixel to be an anchor if the pixel's gradient value is bigger than both of its neighbors by a user-supplied anchor threshold. This is nothing but non-maximal suppression with a user-supplied threshold. To perform this step in CUDA, we make use of the previously computed gradient magnitude and direction maps, which have been stored in the shared memory. Since the kernels use data from the shared memory, the anchors can be computed very fast.

2.2.4. Linking of Edge Anchor Points with Smart Routing in CUDA Kernel

The final step of ED is the linking of anchors by drawing edges between them. Recall that anchors correspond to the peaks of the gradient map. To link consecutive anchors, we simply go from one peak (anchor) to the next by walking over the peaks of the gradient map. This process is guided by the gradient magnitude and direction maps computed in step 2 of the algorithm.

TABLE II. ALGORITHM to WALK LEFT of AN ANCHOR

Symbols used in the algorithm:
row: threadIdx.y; col : threadIdx.x; [row,col]: Thread Index;
G: Gradient map; D: Direction map; E: Edge map;

```
GoLeft(row, col, G, D, E){
    while (G[row, col] > 0 && D[row, col] == HORIZONTAL && E[row, col] !=
                                EDGE){
        E[row, col] = EDGE;        // Mark this pixel as an edgel

        // Look at 3 neighbors to the left & pick the one with the max. gradient value
        if (G[row-1,col-1] > G[row,col-1] && G[row-1,col-1] > G[row+1,col-1]){
            row = row-1; col = col-1;    // Up-Left

        }else if (G[row+1,col-1] > G[row,col-1] && G[row+1,col-1] > G[row-1,col-1]){
            row = row+1; col=col-1;  // Down-Left

        } else {
            col = col-1;            // Straight-Left
        } //end-else
    } //end-while
} //end-GoLeft
```

The algorithm in Table II describes smart routing starting at an anchor (x, y). To perform this anchor linking step, we have implemented a new CUDA kernel. The reason for having a separate kernel for this step is because the gradient magnitude and direction map computation must have finished for all pixels before this step can begin executing. Within this step, each thread checks whether its pixel is an anchor. If not, it terminates; otherwise it runs the anchor linking code given in Table II. Thus, there are many threads performing anchor linking simultaneously. "Fig. 1" shows this step of ED. Red dots in figure correspond to the anchor points obtained in the previous step. Each thread simply starts at an anchor, runs an algorithm similar to the one given in Table II.

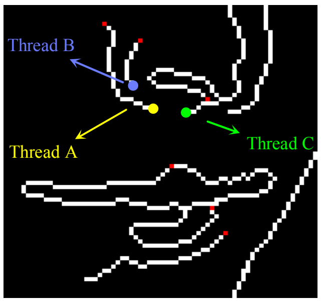

Figure 1. Illustration of Edge Linking.

3. EXPERIMENTAL RESULTS

To measure the performance of our CUDA implementation of ED (to be called CUDA-ED) to a strictly serial CPU implementation (to be called CPU-ED), we run CUDA-ED at four different CUDA cards. CPU-ED was run in a PC having a 3Ghz Intel Pentium 4 CPU and 1GB RAM. Since CPU-ED has already been compared to OpenCV Canny in [8], we will only compare CPU-ED to CUDA-ED in this paper and look at the speedup numbers.

In the following tests, we will use four standard images with different sizes: Specifically, we use "Lena", "Mandril", "Boats" and "Cameraman" images at 64K, 256K, 1M and 16M pixels resolutions.

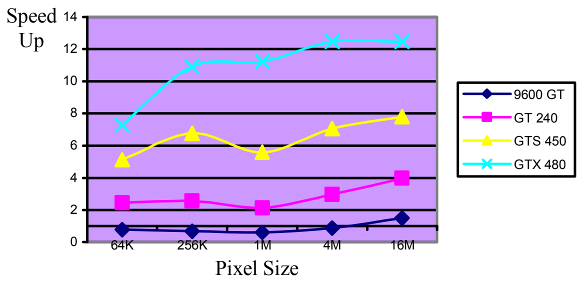

Figure 2. The speedup of CUDA-ED to CPU-ED on Lena image at four different GPU cards as the image size increases.

We see from "Fig. 2" that CUDA-ED runs slower at 9600GT for small image sizes, but runs up to 12 times faster at GTX480 for large image sizes. All-in-all we see that CUDA-ED runs at least 3 times faster and up to 12 times faster than CPU-ED.

"Fig. 3" shows the dissection of the running times of CPU-ED and CUDA-ED on four different images of size 256K pixels at GT240. Kernel1 represents the Gaussian filtering time, Kernel2 represents gradient and direction map computation, and Kernel3 represents the anchor computation and linking times. It is clear from "Fig. 3" that both CPU-ED and CUDA-ED runs real-time for all images. It is also seen that CUDA-ED runs up to 4 times faster than CPU-ED and takes less than 4ms in all images. We also see that memory-to-GPU copy takes a non-negligible amount of time and that the most dominant time in CUDA-ED is the anchor linking step. This is expected since in anchor-linking each thread works on an independent portion of the image, and it is not easy for the GPU card to schedule independent threads in parallel. So, many threads may have to be run serially leading to the observed behavior.

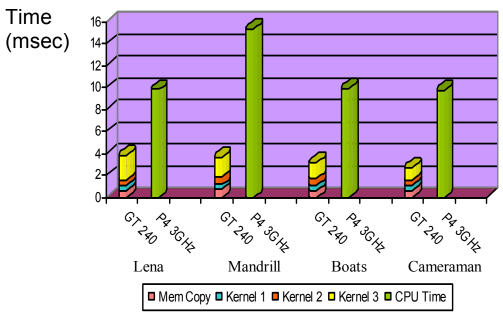

Figure 3. The dissection of the running time of CPU-ED and CUDA-ED on four images of size 256K pixels at GT240.

TABLE III. RUNNING TIMES of CUDA-ED on LENA.

		64 K	256 K	1 M	4 M	16 M
P4 3 Ghz	**Time**	2,41	9,83	34,72	116,27	410,17
9600 GT (64 core)	**Time**	3,11	14,38	57,29	130,79	274,76
	Speed Up	0,77	0,68	0,61	0,89	1,49
GT 240 (96 Core)	**Time**	0,98	3,85	16,19	39,15	103,2
	Speed Up	2,46	2,55	2,14	2,97	3,97
GTS 450 (192 Core)	**Time**	0,47	1,45	6,21	16,47	52,63
	Speed Up	5,13	6,78	5,59	7,06	7,79
GTX 480 (480 Core)	**Time**	0,33	0,9	3,09	9,32	32,88
	Speed Up	7,30	10,92	11,24	12,48	12,47

Table III shows the running times of CUDA-ED and CPU-ED on Lena image as the image size increases along with the speedups obtained. 9600GT is an old CUDA card, and therefore we do not observe a speedup until image size reaches 16M pixels. At newer GPU cards with more cores, CUDA-ED runs really fast. At GTX480 for example, CUDA-ED runs at least 7, up to 12 times faster than CPU-ED. GTX480 is a high-end GPU card and is expensive for a typical end-user. But GT240 is a very common cheap GPU card. Even at this GPU card, CUDA-ED runs at least 2.5, up to 4 times faster than CPU-ED, and takes a mere 1.45ms on a 512x512 image. It clear from "Fig. 4" that CUDA-ED's edgemaps are very high-quality.

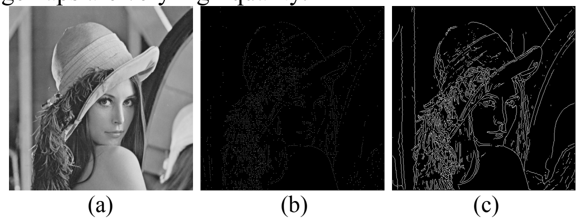

Figure 4. (a) 256K pixels original image. (b) Anchor points kernel output. (c) CUDA-ED's final edgemap.

4. CONCLUSION

This paper presents how our novel edge segment detection algorithm, the Edge Drawing (ED), can be implemented in the CUDA platform. We have presented the implementation details, performed several experiments comparing the performance of CUDA-ED to CPU-ED and have shown that CUDA-ED can run up to 12 times faster than CPU-ED. Although CPU-ED is already a real-time edge segment detector for typical camera input sizes, e.g., 640x480, CUDA-ED would run in amazing speed and would take almost no time. Our observation is that although CUDA GPU cards can perform processing at a very high-speed, memory-to-GPU copy still takes a considerable time.

5. REFERENCES

[1] J. R. Fram and E. S. Deutsch, "On the Quantitative Evaluation of Edge Detection Schemes and Their Comparison With Human Performance," IEEE Trans. Computers, June 1975, vol. 24, no. 6, pp. 616-628.

[2] D. J. Bryant and D. W. Bouldin, "Evaluation of Edge Operators Using Relative and Absolute Grading," Proc. IEEE Computer Society Conf. Pattern Recognition and Image Processing, Chicago, 1979, pp. 138-145.

[3] V. Ramesh and R. M. Haralick, "Performance Characterization of Edge Detectors," SPIE, vol. 1,708, Applications of Artificial Intelligence X: Machine Vision and Robotics, 1992, pp. 252-266.

[4] I. E. Abdou and W. K. Pratt, "Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors," Proc. IEEE, May 1979, vol. 67, no. 5, pp. 753-763.

[5] R. N. Strickland and D. K. Cheng, "Adaptable Edge Quality Metric," Optical Eng., May 1993., vol. 32, no. 5, pp. 944-951.

[6] X. Y. Jiang, A. Hoover, G. Jean-Baptiste, D. Goldgof, K. Bowyer, and H. Bunke, "A Methodology for Evaluating Edge Detection Techniques for Range Images," Proc. Asian Conf. Computer Vision, 1995, pp. 415-419.

[7] T. Kanungo, M. Y. Jaisimha, J. Palmer, and R. M. Haralick, "A Methodology for Quantitative Performance Evaluation of Detection Algorithms," IEEE Trans. Image Processing, Dec. 1995, vol. 4, no. 12, pp. 1,667- 1,674.

[8] C. Topal, C. Akınlar, Y. Genç, "Edge Drawing: An Heuristic Approach to Robust Real-Time Edge Detection", ICPR2010, 23-26 August, 2010.

[9] Y.Luo, R. Duraiswami, "Canny Edge Detection on NVIDIA CUDA", Computer Vision and Pattern Recognition Workshops, CVPRW '08, June 2008, pp. 1-8.

[10] Asano S., Maruyama T., Yamaguchi Y., "Performance comparison of FPGA, GPU and CPU in image processing", Field Programmable Logic and Applications 2009 FPL 2009 International Conference on (2009), pp. 126-131

[11] Datla S., Gidijala N.S.,"Parallelizing Motion JPEG 2000 with CUDA", ICCEE '09, pp. 630-634.

[12] Ronghui C., Eryan Y., Ting L., "Speeding up motion estimation algorithms on CUDA technology", PrimeAsia 2010, pp. 93-96.