Efficient Canny Edge Detection using a GPU

Kohei Ogawa, Yasuaki Ito, and Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima, 739–8527 Japan,

***Abstract*—Recent GPUs, which have many processing units connected with a global memory, can be used for general purpose parallel computation. Users can develop parallel programs running on GPUs using programming architecture called CUDA (Compute Unified Device Architecture). The main contribution of this paper is to implement a Canny edge detection algorithm on CUDA. The experimental result shows that our implementation of Canny edge detection algorithm on CUDA achieves a speedup factor of 61 over a conventional software implementation.**

***Index Terms*—Edge detection, CUDA, GPU, Parallel processing**

I. Introduction

A GPU (Graphics Processing Unit) is a specialized microprocessor that accelerates 3D or 2D graphics operations. Recent GPUs, which have many processing units connected with a global memory, can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [1] is the architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. So, many studies have been devoted to implement parallel algorithms using CUDA [2]–[5].

Canny edge detection algorithm [6], one of the most commonly used image processing algorithms, detects high-quality edges in images. Using this algorithm, we can obtain accurate edges of input images. Essentially, Canny edge detection consists of four steps; **Step 1**: Gaussian filtering, **Step 2**: Sobel filtering, **Step 3**: non-maximum suppression, and **Step 4**: hysteresis thresholding. In Step 1, Gaussian filtering is used to filter out noises in the images by smoothing. This can be done by convolution using a Gaussian filter kernel in Figure 1. In Step 2, Sobel filtering is used to find the edge strength by

0.0625	0.125	0.0625
0.125	0.25	0.125
0.0625	0.125	0.0625

Fig. 1. 3×3 Gaussian filter

d_x filter

−1	0	1
−2	0	2
−1	0	1

d_y filter

−1	−2	−1
0	0	0
1	2	1

Fig. 2. 3×3 d_x and d_y Sobel filters

taking the gradient of the image. For this purpose, a pair of Sobel filter kernels in Figure 2 is used to detect the intensity changes for horizontal direction d_x and the vertical direction d_y. Also, the magnitude M of edge and the gradient direction θ are computed for each pixel by the formulas $M = \sqrt{d_x^2 + d_y^2}$ and $\theta = \tan^{-1}(\frac{d_y}{d_x})$, respectively. In Step 3, non-maximum suppression is used to get thin edges in the image. In this step, local maximum along the gradient direction is detected using M and θ. After that, in Step 4, hysteresis thresholding is used to detect the final edges in image using two thresholding values t_low and t_high. In this step, the pixels are classified into three categories in terms of the value of the magnitude M. If $M \geq t\_high$, the pixel is classified as a *strong edge*. If $t\_low \leq M < t\_high$, the pixel is classified as a *weak edge*. If $M < t\_low$, the pixel is classified as a *non-edge*. Based on this classification, final edge pixels are determined as follows: (1) A strong edge pixel is always an edge pixel. (2) A weak edge pixel is an edge pixel if it is adjacent to an edge pixel. (3) A non-edge pixel is never an edge pixel. Thus, we need to traverse all weak edge pixels starting from strong edge pixels to find all edge pixels. Figure 3 shows the process of Canny edge detection.

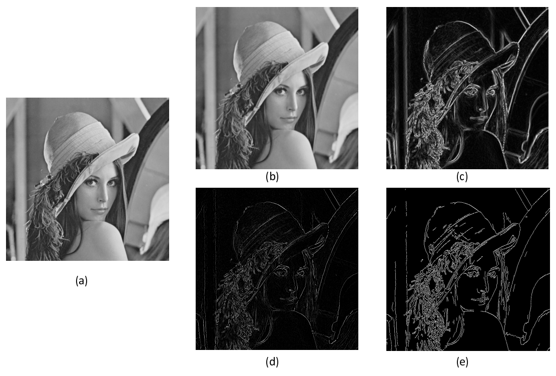

Fig. 3. Canny edge detection process (a) Input image (b) Gaussian filtering (c) Sobel filtering (d) Non-maximum suppression (e) Hysteresis thresholding

In [7], the Canny edge detection was implemented on CUDA. However, their algorithm has flaw. Their algorithm may not traverse all weak edge pixels in Step 4. In their algorithm, the input image is divided into subimages. The weak edges are traversed in each subimage. To traverse the weak edges which cross over other subimages, Step 4 for subimages is performed fixed times. Thus, their algorithm may not obtain the correct results. On the other hand, our method can always traverse all weak edge and obtain correct results.

II. Implementation

Figure 4 shows the CUDA hardware (GPU) architecture [8]. GPU has several *Streaming Multiprocessors* (*SMs*) and a global memory. Each SM has eight processor cores and threads, units of processing on CUDA, run on them in parallel. Also, each SM has a shared memory that can be accessed by its eight processor cores. The global memory whose size is much larger than the shared memory can be accessed by all SMs. However, the access time of the global memory is generally longer. Therefore, shared memories are often used as a cache memory of the global memory.

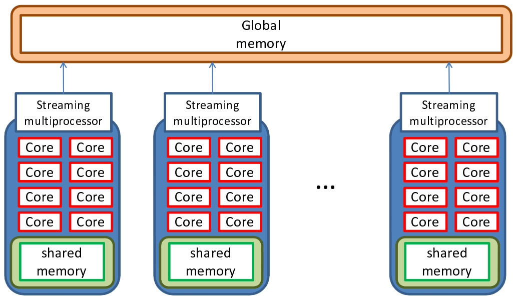

Fig. 4. CUDA hardware (GPU) architecture

In our implementation, shared memories are efficiently used to reduce the number of reading and writing operations for the global memory. However, the input image must be stored in global memory, because the shared memory is too small to store it. The details of our implementation for the Canny edge detection on CUDA are spelled out as follows.

Gaussian and Sobel filtering: The input image is divided into subimages and each of them is assigned to a SM. Each SM stores the assigned subimage into its shared memory and convolutions of Gaussian filter and Sobel filter are computed in parallel. After that, the resulting values of M and θ are written to the global memory.

Non-maximum suppression: Using the resulting values M and θ, each thread checks whether the pixel is local maximum or not. Pixels with a local maximum are recorded in the global memory as candidates of edge pixels.

Hysteresis thresholding: We assign each subimage to a SM and each pixel is processed by the corresponding thread. If the pixel is a strong edge pixel, the thread starts to traverse weak edge pixels adjacent to it. For this purpose, we use a stack for such pixel. First, if the pixel is a strong edge pixel, we check all neighboring pixels. The thread pushes each neighboring pixel into the stack if it is a weak edge pixel. After that, thread pop a pixel from the stack and do the same operation for this pixel. This operation is repeated until the stack becomes empty. In this way, weak edge pixels are traversed starting from a strong edge pixel. Pixels traversed are labeled as a final edge pixel. Figure 5 shows the process of the edge hysteresis thresholding.

III. Experimental Results

We have evaluated the performance of our Canny edge detection algorithm implemented on CUDA. For the purpose of comparison, we have also implemented a conventional software approach running on a single CPU on a PC. We have used Tesla C1060 with 240 cores running in 1.3GHz and 4GB memory. To evaluate a conventional software approach, we have used a PC server with Intel Xeon E5540 running in 2.0GHz and 6GB memory.

Table I shows the performance of CUDA and a single CPU. The size of input image is 10240×10240. The data transfer

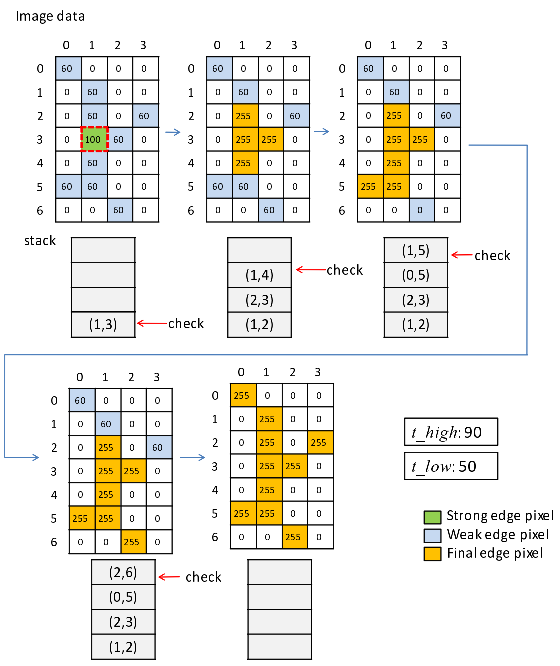

Fig. 5. Process of Hysteresis Thresholding

TABLE I
Performance of GPU and CPU

	CPU	GPU (without data transfer)	GPU (with data transfer)
Time [ms]	22289.45	364.389	444.29

is necessary for CUDA. CUDA implementation achieved approximate 50 times speedup. If the data transfer time is not included, CUDA implementation achieved approximate 61 times speedup.

IV. Conclusion

In this paper, we implemented the Canny edge detection using CUDA. Comparing to the performance of CPU system, CUDA can achieve 50 times speed-up.

References

[1] NVIDIA, "CUDA ZONE," http://www.nvidia.com/page/home.html.
[2] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, October 2008, pp. 171–175.
[3] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell, "A parallel implementation of k-means clustering on GPUs," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, July 2008, pp. 340–345.
[4] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, 2007, pp. 197–208.
[5] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proceedings of International Parallel and Distributed Processing Symposium*, 2010.
[6] J. F. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, November 1986.
[7] Y. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *Proceedings of the Workshop on Computer Vision on GPUS*, June 2008, pp. 1–8.
[8] NVIDIA, *NVIDIA CUDA Programming Guide*, July 2009.