Canny Edge Detection on NVIDIA CUDA

Yuancheng "Mike" Luo and Ramani Duraiswami
Perceptual Interfaces and Reality Laboratory
Computer Science & UMIACS, University of Maryland, College Park
{yluo1@,ramani@umiacs.}umd.edu
http://www.wam.umd.edu/~yluo1/canny.htm http://www.umiacs.umd.edu/~ramani

Abstract

The Canny edge detector is a very popular and effective edge feature detector that is used as a pre-processing step in many computer vision algorithms. It is a multi-step detector which performs smoothing and filtering, non-maxima suppression, followed by a connected-component analysis stage to detect "true" edges, while suppressing "false" non edge filter responses. While there have been previous (partial) implementations of the Canny and other edge detectors on GPUs, they have been focussed on the old style GPGPU computing with programming using graphical application layers. Using the more programmer friendly CUDA framework, we are able to implement the entire Canny algorithm. Details are presented along with a comparison with CPU implementations. We also integrate our detector in to MATLAB, a popular interactive simulation package often used by researchers. The source code will be made available as open source.

1. Introduction

A recent hardware trend, with origin in the gaming and graphics industries, is the development of highly capable data-parallel processors. These "graphics" processors were originally meant to rapidly create two-dimensional images from 3D models and textures. They are architected to quickly perform all the operations in the graphics pipeline. In 2007, while the fastest Intel CPU could only achieve ~20 Gflops, GPUs had speeds that are more than an order of magnitude higher. Of course, the GPU performs highly specialized tasks, while the CPU is a general purpose processor. Fig. 1 shows the relative abilities of GPUs and CPUs (on separate benchmarks) till 2007. The trends reported are expected to continue for the next few years.

While GPUs were originally intended for specialized graphics operations, researchers in other domains, including computer vision, wanted to use their capabilities to solve their problems faster. This has been a vigorous area of research over the last five years. In the first few years the approach was to "fool" the GPU in to thinking that it was performing graphics. In this approach, termed general purpose GPU computing or GPGPU, the image processing and computer vision algorithms were mapped in to a graphical framework (calling OpenGL or DirectX functions). Unfortunately, the need to translate regular programs in to graphics metaphors has meant that only a small group of dedicated researchers with graphics knowledge used the power of the GPUs for non-graphical computing. The computer vision library OpenVidia [7] is an application of this type, as is GpuCV [12].

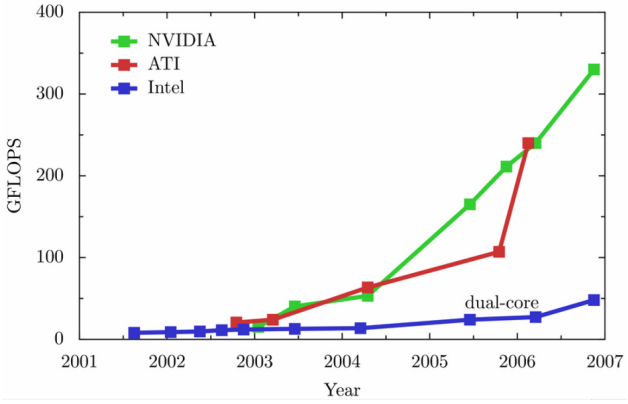

Figure 1. GPU and CPU growth in speed over the last 6 years (from [1]).

1.1. Coprocessor Programming Frameworks

Since early 2007, GPU manufacturers have begun to market them as *compute coprocessors*, and this is the way we consider them. The manufacturers have opened up the functionality of these coprocessors using common programming languages, and users no longer need to have knowledge of graphics concepts. This is especially useful when programming items that are not directly related to items that are in the graphics pipeline. While AMD/ATI has also recently released a new class of GPUs and a program-

ming methodology, the Firestream [10]; and the Sony-IBM-Toshiba cell processor [11] has also been used for general purpose programming for the last couple of years, we are more familiar with the version released by NVIDIA. It consists of a programming model (Compute Unified Device Architecture or CUDA) and a compiler that supports the *C* language with GPU specific extensions for local, shared and global memory, texture memory, and multithreaded programming. The ability to program in a more "native" fashion means that more complex algorithms and data structures can be more easily implemented in this framework.

The NVIDIA G80 GPU, the one on which we developed our software, is the 2007-2008 generation of the NVIDIA GPU, and has also been released as the Tesla compute coprocessor. It consists of a set of multiprocessors (16 on our GeForce 8800GTX), each composed of 8 processors. All multiprocessors talk to a global device memory, which in the case of our GPU is 768 MB, but can be as large as 1.5 GB for more recently released GPUs/coprocessors. The 8 processors in each multiprocessor share 16 kB local read-write "shared" memory, a local set of 8192 registers, and a constant memory of 64 kB over all multiprocessors, of which 8 kB can be cached locally at one multiprocessor.

The CUDA model is supposed to be extended over the next few generations of processors, making investment of effort on programming it worthwhile, an important consideration for researchers who have spent significant time on short-lived parallel architectures in the past. Under CUDA the GPU is a compute device that is a highly multithreaded coprocessor. A thread block is a batch of threads that executes on a multiprocessor that have access to its local memory. They perform their computations and become idle when they reach a synchronization point, waiting for other threads in the block to reach that point. Each thread is identified by its thread ID (one, two or three indices). The choice of 1,2 or 3D index layout is used to map the different pieces of data to the thread. The programmer writes data parallel code, which executes the same instructions on different data, though some customization of each thread is possible based on different behaviors depending on the value of the thread indices.

To achieve efficiency on the GPU, algorithm designers must account for the substantially higher cost (two orders of magnitude higher) to access fresh data from the GPU main memory. This penalty is paid for the first data access, though additional contiguous data in the main memory can be accessed cheaply after this first penalty is paid. An application that achieves such efficient reads and writes to contiguous memory is said to be *coalesced*. Thus programming on the nonuniform memory architecture of the GPU requires that each of the operations be defined in a way that ensures that main memory access (reads and writes) are minimized and each piece of data that is read has significant computation performed on it; read/writes should be coalesced as far as possible when they occur.

1.2. Present contribution

In this paper we focus on a GPU implementation of the Canny edge detector [3]. This algorithm has remained a standard in edge finding techniques over the years. Applications of edge detection include their use as features in vision algorithms, their use to improve the appearance of displayed objects, in image coding and others too numerous to discuss. Many implementations of the Canny algorithm have been made on various platforms in the past, including in the earlier GPGPU. A partial Canny edge filter for the GPU has been presented in OpenVIDIA using NVIDIA's Cg shader language [7]. This implementation, however, does not include the hysteresis labeling connected component part. Neoh and Hazanchuk [5] have presented an implementation on the Field-programmable gate array (FPGA), again without the connected components part. In both cases, the reason for the lack of the connected component is related to the need for non-local memory, which causes significant slowdowns.

On the CPU several versions are available, with varying efficiency. Matlab is typical of most CPU implementations, and has a library function to find edges using the Canny technique. Recently, the Intel Open Computer Vision Performance Library (OpenCV) [6] contains a CPU assembly optimized version of the Canny detector, capable of multithreaded multi-core operation, and this is the benchmark against which we will compare our algorithm. On modern multicore systems this is much faster than any other implementation by an order of magnitude, including Matlab's image processing toolkit version.

In the following sections we present a brief introduction to the Canny algorithm, followed by a discussion of how various parts of it were mapped to the GPU architecture. We next describe its (relatively straightforward) implementation as a command under Matlab. We then present benchmark computations that show the performance that is achieved for various standard images. Significant speedups are reported against the Matlab version (~30 to 80 times), while an approximately three to five fold improvement against the Intel OpenCV version running on a state of the art Intel processor. Our results clearly demonstrate the programmer friendly approach in CUDA, allows the layout of relatively complex algorithms and data structures in a way that allows the efficiency of these processors to be exploited.

2. The Canny Algorithm

Canny [3] defined optimal edge finding as a set of criteria that maximize the probability of detecting true edges while minimizing the probability of false edges. He found that the zero-crossings of the second directional derivative of a smoothed image were a reasonable measurement of ac-

tual edges. To smooth the image, the Canny edge detector uses Gaussian convolution. Next, the image is convolved with a 2D first derivative operator to determine regions of sharp changes in intensities. The gradient magnitude and direction at each pixel are calculated in this step. Note that the maxima and minima of the first derivative gradient are the same as the zero-crossings of the second directional derivative. Only the maxima crossings are of interest because these pixels represent the areas of the sharpest intensity changes in the image [4]. These zero-crossings are the ridge pixels that represent the set of possible edges. All other pixels are considered non-ridge and subsequently suppressed. Finally, a two-threshold technique or hysteresis is performed along the ridge pixels to determine the final set of edges.

Instead of using a single threshold value for filtering ridge pixels, the Canny algorithm implements a connected components analysis technique based on a hysteresis thresholding heuristic. This step uses two thresholds, $t1, t2$ where $t1 > t2$, to partition the ridge pixels into edges/non-edges. Pixels with gradient magnitudes above $t1$ are classified as definite edges. Pixels between $t2$ and $t1$ are classified as potential edges. Pixels under $t2$ are classified as non-edges. Next, all potential edges that can be traced back to a definite edge via adjacent potential edges are also marked as definite edges. The process solves some of the issues associated with edge streaking and discontinuity in the results achieved by simple detectors by identifying strong edges while accounting for comparatively weaker ones.

2.1. Algorithm costs on a CPU

Given an $M \times N$ image, an examination of a standard Canny edge detector implementation using a Gaussian filter with a 1D window size W pixels, a Sobel operator with a 1D aperture of Y pixels, and that yields K candidate edges on a serial processing machine shows the following run-times:

Functions	Operations
Smoothing w. Separable Filters	$2WMN$
Gradients via 1-D Sobel	$2YMN$
Non-maximum Suppression	$2MN$
Hysteresis comparisons	$< MN$

Two special functions are needed, which add to the costs:

Gradient Magnitude	MN (Square root function)
Gradient Direction	MN (Arc-tangent function)

3. GPU Algorithm

The GPU hardware allows for fast performance of pixel-wise operations required for a Canny edge detector implementation. Many steps such as gradient finding, convolution, and non-maximum suppression can be performed in parallel on a pixel-wise level. Our implementation is part of an application that is designed to process 24-bit RGB images after performing a grayscale conversion, while satisfying Canny's criteria for optimal edge finding. While this is basically the Canny detector running on a per color basis, the memory pattern makes it unique enough. A version that is meant to operate on 8-bit grayscale images, or higher bit-depth grayscale images can be easily adapted from our version. The primary goal is to get an optimized parallel algorithm that utilizes the NVIDIA N80 SIMD architecture model and processing power of the GPU. We first use a separable filter algorithm, similar to the one supplied with the CUDA toolkit to implement the Gaussian smoothing [9].

Our implementation has a host processing component and a kernel (GPU) component. The host component consists of a fast retrieval and decompression of video or image data into accessible arrays for kernel processing. We used OpenCV, a third party computer vision library, to retrieve the data. The format of our data array is in 24 bit row major order with 8-bit width color channels. The data is then loaded into GPU global memory. While, the transfer is not a significant part of the access, in more dynamic situations, e.g., where an incoming stream has to be processed online, a direct transfer of data in to the GPU global memory might be beneficial, and NVIDIA forums show that this feature for the CUDA drivers has been requested by many users. The kernel portion performs the necessary steps of the Canny edge detector. Much of the implementation is optimized for the CUDA architecture although some aspects are not yet as efficient as we would like.

3.1. Efficient CUDA implementation

The target GPU used for testing is the NVIDIA 8800 GTX, which includes 128 SIMD stream processors working together in groups of 8 to form 16 multiprocessors. Much of the code as a result is based on optimizing the memory usage, alignment, and size properties of the GPU architecture. Additionally, we set additional conditions on the image data beforehand to be processed. The image data must be in a linear 24 bit per pixel format. Every "pixel" must be constrained to 3 channels of 8bit width each. The suggested color space is RGB since the Canny algorithm is best suited for grayscale images converted from RGB space. The image width and height must be of a multiple of 16 pixels to fit global memory access alignment properties.

All of our kernel threads belong to groups of 256 or 16x16 threads per block. Each thread generally correlates to a single pixel for processing since many of the Canny algorithm operates on a pixel-wise basis.

CUDA global memory loads: To process data from the GPU global memory space, the kernel threads must meet the memory and alignment requirements for loading data. Given the 24-bit per pixel data format, we cannot have consecutive threads load their pixel data from global memory as that would fail to meet alignment standards. To have coalesced memory access, each half-warp or 16 threads must access the memory address of $HalfWarpBaseAddress + N$ [2]. Furthermore, single

thread accesses are limited to reading 32-bit, 64-bit, or 128-bit words from a single access making our 24-bit per pixel format difficult.

Normally, a 16-thread half-warp processes 24*16=384 bits of data assuming a one-to-one correspondence between thread and pixel. However, a one-to-one correspondence with our data format does not meet memory alignment requirements. The solution was to have the first 12 threads of each row in a block load consecutive 32-bit unsigned integers per iteration and dump 8-bit chunks of each 32-bit load into shared memory space. Segmenting the 8-bit chunks from the unsigned integers was done using fast bit-wise operations available on the GPU. Subsequent stages of processing were done after conversion into regular floats for efficient floating point operations.

```
__shared__ int smem [BLOCK_HEIGHT][BLOCK_WIDTH][3];
unsigned char tx = threadIdx.x;
unsigned char ty = threadIdx.y;
unsigned char bx = blockIdx.x;
unsigned short int x = BLOCK_WIDTH*bx+tx;
unsigned short int y =BLOCK_HEIGHT*blockIdx.y+ty;
unsigned int ref = 3 * (iWidth * y + x);
unsigned int globalMemAddress = 3*(iWidth*y
+BLOCK_WIDTH*bx)+tx*4;//Every halfwarp
should be multiple of 16*sizeof(type)
unsigned short int
halfwarpSmemAddressPlusOffset = 3 + tx * 4;
__syncthreads();
if(tx<12){//As long as every halfwarp
thread is aligned and subsequent threads
read from halfwarpthread+1 address of type
with size 4, coalesce will occur
  unsigned int iDataVal = *((int*) (iData +
  globalMemAddress));//Single 4byte read
  //Segment and put into shared memory
  *(&smem[ty][0][0]+halfwarpSmemAddressPlus
  Offset)=(iDataVal)&0x000000FF;
  *(&smem[ty][0][0]+halfwarpSmemAddressPlus
  Offset+1)=(iDataVal>>8)&0x000000FF;
  *(&smem[ty][0][0]+halfwarpSmemAddressPlus
  Offset+2)=(iDataVal>>16)&0x000000FF;
  *(&smem[ty][0][0]+halfwarpSmemAddressPlus
  Offset+3)=(iDataVal>>24)&0x000000FF;}
```

Figure 2. Code to read in RGB image in to shared memory.

The 12 threads still load the equivalent of 12*32=384 bits of global memory to be used per half-warp at the cost of underutilizing 25% of the threads during the read. However, this was an acceptable cost since no bits were "redundantly" read from global memory while achieving memory coalesced reads.

Gaussian and Sobel Filtering: It is important to note that Sobel and Gaussian filters are separable functions. Generally, a non-separable filter of window size $M \times M$ computes M^2 operations per pixel, whereas for a separable filters are used, the cost would be reduced to computing $M+M = 2M$ operations. This is a two step process where the intermediate results from the first separable convolution is stored and then convolved with the second separable filter to produce the output.

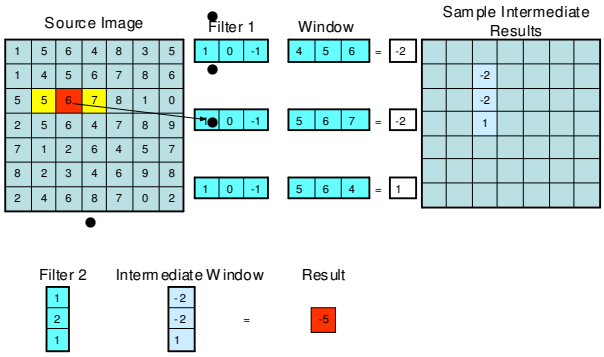

Figure 3. Separable filter convolution.

The source image data is accessed through threads that load a corresponding pixel into the block's shared memory space. Non-separated $M \times M$ filters require each thread block to load an additional apron of $\lfloor M/2 \rfloor$pixels wide. The total number of pixels loaded into the apron is 4*16* $\lfloor M/2 \rfloor$+4* $2\lfloor M/2 \rfloor$. Apron pixels are necessary because convolutions near the edge of a thread block will have to access pixels normally loaded by adjacent thread blocks. Since shared memory is local to individual thread blocks, each block loads its own set of apron pixels before processing. For a separated $M \times M$ filter, the apron does not consist of corners since each filter would only need pixels along a single dimension. The total number of pixels loaded is 4*16*$\lfloor M/2 \rfloor$, which is a speedup compared to the non-separated filters.For a Sobel filter of window size 3,

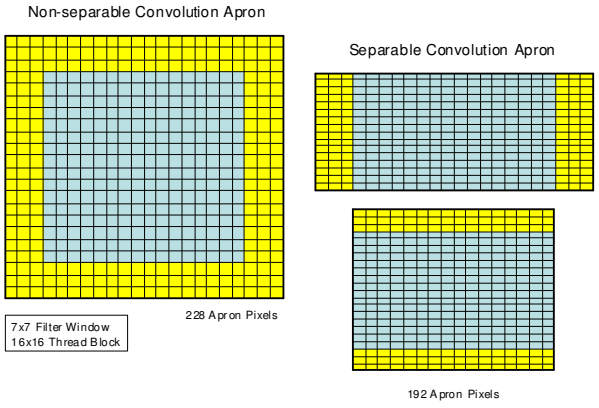

Figure 4. "Apron" pixels are used in each block to accomodate the filters.

the speedup is moderate. For larger filters such as the 2D Gaussian where the window width may be larger than the thread block, the speedups can be noticeable.

Gradient Computations: We use two functions to

convolve the separable Sobel filters of aperture size 3, with the source image and compute the gradients G_x and G_y at each pixel. The horizontal pass convolves the source image across the columns. The necessary apron pixels cannot be all memory coalesced from global memory because they do not belong to the same half-warp base address as the center group. The use of some data structure could alleviate this, but we have not attempted it yet. The vertical pass convolves the source image down the columns. The apron pixels here are memory coalesced because their addresses are contiguous and property aligned above or below the center group (See Fig. 4). To calculate the gradient strength, we take the norm of the two gradients, while the gradient direction is found by $\theta = \arctan(G_y/G_x)$. The direction is quantized to point to one of the neighboring pixels with angles $\{\pi/8 + k\pi/4\}$.

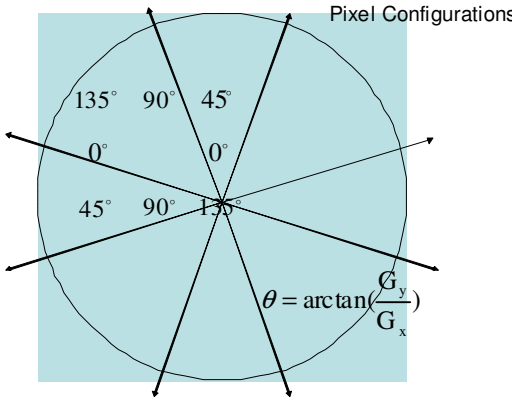

Figure 5. The gradient direction computations are quantized to eight directions corresponding to neighbors.

CUDA non-maximum suppression: Once the gradient magnitude and direction data have been found, a new function checks each pixel for the ridge criteria and suppresses all non-ridge pixels. Ridge pixels are defined as the pixels with gradient magnitudes greater than both of its adjacent pixels in the gradient direction. Non-ridge pixels are suppressed by having their gradient magnitude set to 0. The function also requires a 1 pixel wide apron around each thread block since pixels along the perimeter have directional configurations extending outside of the normal pixel group range of the thread block.

CUDA hysteresis and connected components: The hysteresis and connected components algorithms are the final steps to producing edges. For the hysteresis portion, a high threshold and low threshold are specified by the user beforehand. The process begins by marking all pixels with gradient magnitudes over the high threshold as a "discovered" definite edge. These pixels are placed into a queue and become starting points for a breadth first search (BFS) algorithm. We run the BFS by iterating through the queue

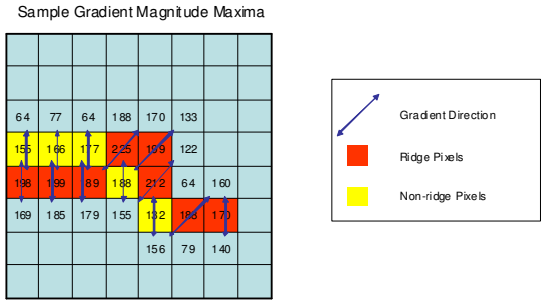

Figure 6. Non-maximum suppression used to find ridge pixels.

of pixels, process each pixel, and remove it from the queue until it is empty.

All adjacent pixels (one of the 8 neighbors) are treated as connected nodes to the current pixel. The criteria for adding new pixels to the queue follows that an adjacent pixel that has not been previously discovered and has a gradient magnitude greater than the low threshold. Adjacent pixels that meet the criteria are subsequently added to the BFS queue. Every adjacent pixel is also marked as discovered once it is checked against the criteria. After all adjacent pixels are checked, the processed pixel is discarded from the queue. The BFS terminates when the queue is empty.

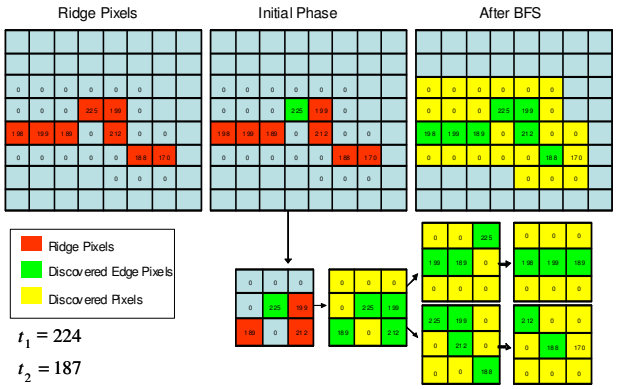

Figure 7. Generalized hysteresis and connected components algorithm using breadth first search. Pixels over the $t1$ threshold ($t1 = 224$,here) are added to the queue. BFS iterates through undiscovered adjacent pixels and adds to queue if pixel value is over $t2$ ($t2 = 187$, here).

Our algorithm is similar to the generalized BFS approach but adheres to the specific constraints of the CUDA architecture. Each thread block processes a separate set of BFS on a group of pixels from the image. A group of pixels is defined as the set of pixels aligned with its corresponding thread block plus a one pixel apron around the block's perimeter. The algorithm is divided into three stages: *Preprocessing*, *BFS*, and *write-back*. In the preprocessing stage, each thread checks if its corresponding

pixel value has a gradient magnitude that is positive or non-positive. We denote positive values as unprocessed pixels and non-positive values as processed pixels. Unprocessed pixels are marked in shared memory as definite, potential, or non-edge if they fall greater, between, or less than the high and low thresholds. We designate these edge states in terms of non-positive values (-2, -1, 0) such that they can be uniquely identified when stored in the same gradient magnitude space.

In the BFS stage, each thread possesses its own queue. Each thread initially checks if its corresponding pixel is a potential edge. If a potential edge pixel is adjacent to one or more definite edge pixels, it is added to the thread's BFS queue. Threads with non-empty queues will run a BFS that connects all adjacent pixels classified as potential edge states inside the thread block's pixel group.

When all threads within a thread block have empty queues, we write the edge states of all non-apron[1] edges in shared memory back into the gradient magnitude space in global memory. After the algorithm terminates, subsequent calls of our algorithm will allow for pixels between adjacent thread blocks to connect during the preprocessing stage. The one pixel apron that is reloaded into shared memory every function call contains the updated edge states of pixels from adjacent thread blocks that were processed in previous iterations. This allows for new pixels with previously potential edge states to be processed at every subsequent function call.

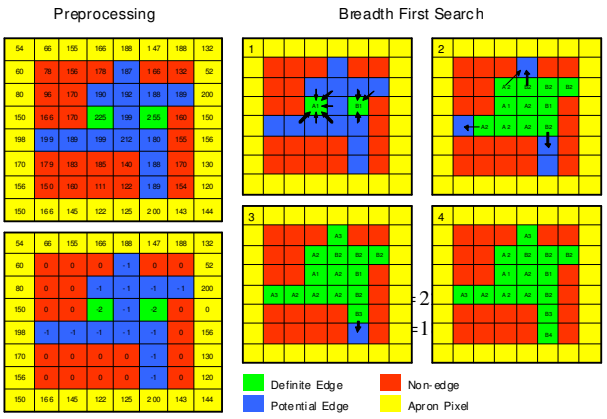

Figure 8. CUDA hysteresis and connected components with breadth first search. Each thread performs a BFS on a group of pixels. Pixel visited/processed states are tracked in shared memory and accessible to all threads in the thread block.

The main issue of hysteresis and connected components in CUDA is the necessity for inter-thread block communication. Although threads within a thread block can be synchronized, threads in different blocks are not. This is problematic for a connected components algorithm when adjacent pixels belonging to adjacent thread blocks should be connected but cannot because of the locality of threads. As a result, a multi-pass approach solution is necessary. For later testing purposes, we call the function four times per iteration.

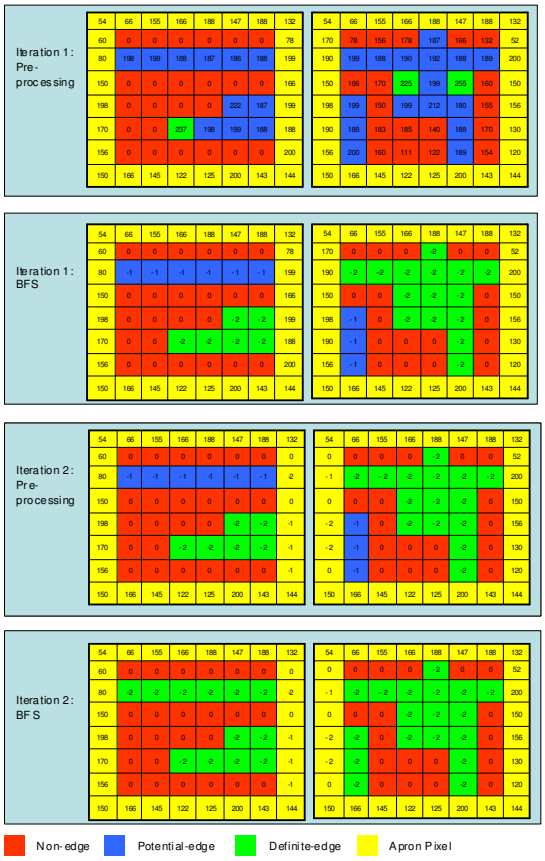

Figure 9. Multi-pass hysteresis and connected components of two adjacent thread blocks. After the first iteration, two pixel chains remain disconnected because two thread blocks performed processing. After the second iteration, the chains connect because of updated apron pixels.

3.2. Matlab interface

Many vision researchers prefer to develop and prototype their algorithms in the Matlab environment. While Matlab is primarily an interpreted environment, and consequently performance can be poorer than in a compiled language, Matlab is extensible and can call compiled "mex" functions. We provided an interface to a function callable from Matlab via the command `C=CudaEdgeRGB(A,thresL,thresH,hyster)`, and a corresponding function for grayscale images.

The Matlab version was essentially a stripped down ver-

[1] Since apron pixels overlap into the address space of other pixel groups and the G80 series does not support atomic writes to global memory.

sion of the original program. It was compiled with only one .cu file since none of the OpenCV, OpenGL (for display), and other interfaces were needed. Care was needed to account for the fact that the arrays are column-major order as opposed to *C*'s row major order.

4. Results

For testing, we compared our GPU Canny implementation with an assembly optimized CPU implementation in the OpenCV library, as well as against the Matlab toolbox implementation. While CUDA speedups can be significant against naive code, hand optimized code taking advantage of special hardware and multimedia processing instructions (SSE) can achieve competitive performance. Our results for both tests were performed on the standard "Lena" and "Mandrill" images.

OpenCV Comparison: The absolute runtimes of the two algorithms were recorded and averaged over 1000 iterations per test. Next, a functional breakdown of runtimes of our implementation is recorded using NVIDIA's cudaProfiler. This data set determines which portions of the implementation benefited the most from the CUDA design. For specificity the hardware used is given below:

- CPU:Intel Core2 CPU 6600 @ 2.40 GHz, 2 GB RAM
- GPU: NVIDIA GeForce 8800 GTX, 768 MB global memory (not overclocked)

Image Size	CUDA (ms)	OpenCV	Speedup
256× 256	1.82	3.06	1.681
512× 512	3.40	8.28	2.435294
1024× 1024	10.92	28.92	2.648352
2048× 2048	31.46	105.55	3.353465
3936× 3936	96.62	374.36	3.874560

Results from "Lena" indicate that fewer edges in the source image exhibit better performance in terms of relative speedup between the GPU and CPU algorithms. Absolute runtime of both CPU and GPU algorithms increases proportionally with image dimension NxN. Relative speedup is also linear, which is an indicating that the GPU implementation is memory bound rather than operation bound. However, the functional runtime data indicates otherwise.

The hysteresis component of the implementation occupies over 75% of the total runtime. This is not unexpected because the hysteresis function takes a multi-pass approach to connect edges between thread block. For testing, the hysteresis function is called four times per iteration and therefore occupies up to 18% of the total runtime per call. The graphical results show that after four iterations the hysteresis sufficiently identified most of the edges in both the Mandrill and Lena images. A higher number of passes showed limited improvements to the final edge map and would occupy more processing time. Note, that the OpenVidia algorithm does not have the hysteresis, and if we turn this part off, we will get a 4 fold improvement.

Figure 10. Results on the image Lena.

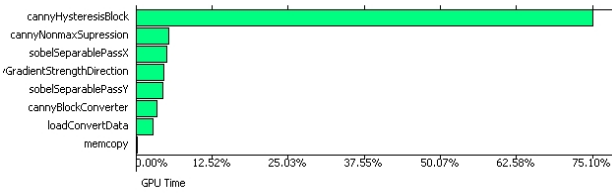

Figure 11. CUDA profiler output for "Lena."

Image Size	CUDA (ms)	OpenCV	Speedup
256x240	1.63	3.92	2.405
512x496	3.60	12.25	3.403
1024x992	12.93	42.70	3.302
2048x1968	31.46	142.70	2.839
4096x3936	153.74	454.60	2.957

Results from the "Mandrill" test shows a moderate increase in relative speedup as image sizes increases. The speedup for higher image resolutions reaches an early plateau at image size 1024x992. This is due to the high edge count in larger image that causes more threads in the hysteresis function to serialize.

The profiler data shows a similar breakdown of functional runtimes with the previous test. Comparing the two tests, the CUDA function performed better for low edge count images in terms of both absolute runtimes and relative speedup with its CPU counterpart.

Matlab: The performance of the Matlab version (M) and the Matlab CUDA (MC) version for the "Lena" and

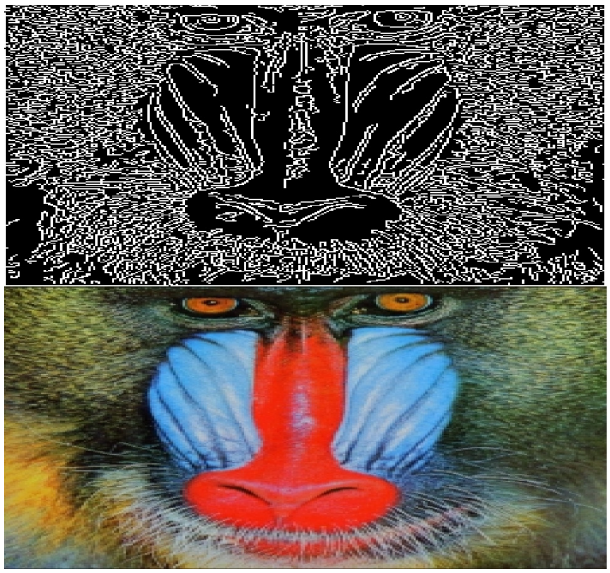

Figure 12. Results on image "Mandrill"

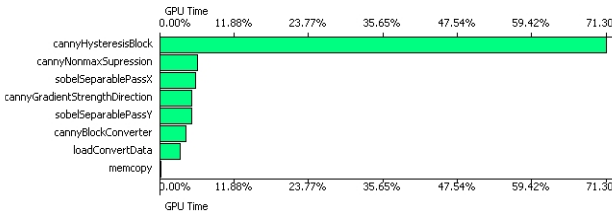

Figure 13. CUDA profiler output on "Mandrill"

"Mandrill" images respectively, at different resolutions are shown below. A significant speedup is observed against the toolbox function. For Matlab, we also developed an 8 bit version for grayscale images, which shows similar processing times (about 10% faster).

$N \times M$	MC (ms)	M (ms)	$N \times M$	MC (ms)	M (ms)
256x256	4.7	102	240x256	4.4	103
512x512	8.6	408	496x512	8.6	429
1024x1024	22.7	1706	992x1024	24.9	1827
2048x2048	75.4	6726	1968x2048	92.1	7065
3936x3936	227.7	26020	3936x4096	309	28388

Canny results from the OpenCV, Matlab, and our implementation all produced the stronger edges in the images. However, results were not identical due to possible differences in implementation. The Matlab edge function uses a derivative of a 2D Gaussian of a variable window size to find the gradients whereas we use a Sobel filter of width 3.

5. Conclusions

We have demonstrated a version of the complete Canny edge detector under CUDA, including all stages of the algorithms. A significant speedup against straight forward CPU functions, but a moderate improvement against multi-core multi-threaded CPU functions taking advantage of special instructions was seen. The implementation speed is dominated by the hysteresis step (which was not implemented in previous GPU versions). If this postprocessing step is not needed the algorithm can be much faster (by a factor of four). We should emphasize that the algorithms used here could be made more efficient, and further speedups should be possible using more sophisticated component data parallel algorithms. Our experience shows that using CUDA one can move complex image processing algorithms to the GPU. The software is available from the 1st author's site.

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26:80–113, 2007. 1

[2] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, V. 1.0, 06/01/2007. 3

[3] J.F. Canny, A computational approach to edge detection. IEEE Trans pattern Analysis and machine Intelligence, 8: 679-698, Nov 1986. 2

[4] D. Marr; E. Hildreth. Theory of Edge Detection. Proceedings of the Royal Society of London. Series B, Biological Sciences, 207:187-217, 1980. 3

[5] Hong Shan Neoh, Asher Hazanchuk Adaptive Edge Detection for Real-Time Video Processing using FPGAs, Application notes, Altera Corporation, 2005, www.altera.com 2

[6] OpenCV computer vision library. intel.com/technology/computing/opencv 2

[7] J. Fung. Computer Vision on the GPU, chapter 40, pages 649-665, in GPU Gems 2, edited by M. Pharr Addison Wesley, 2005. openvidia.sourceforge.net 1, 2

[8] J. Fung and S. Mann. Computer Vision Signal Processing on Graphics Processing Units. Proc. IEEE ICASSP, pp. V-93 - V-96, 2004.

[9] V. Podlozhnyuk, Image Convolution with CUDA. NVIDIA Corporation white paper, June 2097. 3

[10] http://ati.amd.com/products/streamprocessor/specs.html 2

[11] http://en.wikipedia.org/wiki/Cell_microprocessor 2

[12] J.-P Farrugia,.P. Horain, E. Guehenneux and Y. Alusse. GPUCV: A Framework for Image Processing Acceleration with Graphics Processors Proc. IEEE ICME, 585 - 588, 2006.

1