# Efficient implementation of Canny Edge Detection Filter for ITK using CUDA

Luis H.A. Lourenço, Daniel Weingaertner and Eduardo Todt
Vision, Robotics and Image Research Group (VRI)
Informatics Department, Universidade Federal do Paraná
Curitiba, Brazil
Email: {danielw,todt}@inf.ufpr.br

***Abstract*—This work presents an efficient CUDA implementation of the Canny edge detection Filter for the Insight Segmentation and Registration Toolkit (ITK). The algorithm is tested on three generations of NVidia GPGPUs, showing performance gains of $3.6$ to $50$ times when compared to the standard ITK Canny running on two CPU models. The CUDA-enabled Canny is also compared to a more efficient Canny implementation from the OpenCV library. Examples of coding strategies to avoid warp serialization in CUDA are shown on a smart implementation of the Sobel filter, as well as on other algorithms.**

## I. Introduction

Edge detection is among the most used procedures in image pre-processing, having applications in fields such as computer vision and medical image analysis. Enhancing efficiency of a commonly used filter, as the Canny edge detector [1], leads to improved performance in the whole image processing workflow.

Parallelizing image processing algorithms has received considerable research attention, and the General Purpose Graphic Processing Unit (GPGPU), emerging as a powerful and accessible parallel computing platform, has allowed for significant speedups for many image processing tasks [2]–[4]. Studies of GPGPU accelerated Canny edge detection have been presented [5]–[7]. These works achieve reasonable speedups and provide the foundations for this work. Similarities and improvements will be highlighted on the next sessions.

Two major contributions of this paper are: 1) the development of an efficient CUDA-based Sobel algorithm to compute edge magnitude and direction, avoiding conditional expressions and reducing memory access when compared to [5]; and 2) the implementation of an efficient Canny edge detection algorithm, using second order derivatives (instead of Sobel [5]) and a hybrid CPU-GPU approach for the final hysteresis thresholding step, that is independent of the image size (limited to the GPU memory capacity).

The remaining of the paper is organized as follows: In Section II related works are shortly reviewed. In Section III we present our Canny implementation[1] and some efficient programming examples for CUDA. Section IV defines the test methodology, and in Section V experimental results are presented to demonstrate the performance of the algorithms.

[1]The developed algorithms' source code and test images are available at http://web.inf.ufpr.br/vri/alumni/2011-LuisLourenco

## II. Background

### A. NVidia Compute Unified Device Architecture

Developing optimized programs using CUDA demands a good understanding of the GPGPU architecture abstraction and programming paradigm. The creation, scheduling and completion of all threads in CUDA are controlled by the GPU hardware [8]. The threads are organized hierarchically into grids of thread blocks. Each active block is divided into groups of threads called warps. A warp has 32 threads and is scheduled with other warps in a multiprocessor. Each multiprocessor has thread processors and an execution controller.

All thread processors in a warp execute the same instruction simultaneously, so that the maximum efficiency is achieved when all 32 threads in a warp have the same execution flow. Any flow control instruction (**if**, **switch**, **do**, **for**, **while**) might increase the execution time by dividing the warp into divergent execution flows. When this happens, the warp is divided into groups of threads with the same execution flow, and the execution of the thread groups of a warp is serialized by the multiprocessor. Warps that have their execution serialized are called divergent warps.

Avoiding divergent warps is one of the main programming difficulties in CUDA. During the implementation, this was one of the main concerns, and resulted in alternative algorithms to evaluate many conditional expressions, which are discussed on this paper.

The CUDA-compatible GPGPU architectures also offer a hierarchical set of memories. The global memory is the biggest, slowest, and is addressable by all threads. It has two read-only cache memories that can be used to optimize access: the Texture and Constant caches. Each multiprocessor also has an additional memory shared by all threads: the shared memory, and each thread processor in a multiprocessor owns a register set.

Hiding memory access latency is possible by using the shared memory as a cache [9], since it is almost as fast as the processor registers. Texture and Constant caches are also efficient to optimize global memory access, because of their prefetch mechanism. Therefore this work used linear Texture cache for almost all read-only global memory accesses.

### B. The Insight Segmentation and Registration Toolkit

The Insight Segmentation and Registration Toolkit (ITK) is an open-source, cross-platform system that provides developers with an extensive suite of software tools for image analysis. Largely used on medical image processing, ITK's creation was funded in 1999 by the US National Library of Medicine of the National Institutes of Health and developed by the Insight Software Consortium [10].

ITK's architecture is based on workflows, where data objects (images) are processed by a sequence of interconnected filters. ITK uses object oriented programming concepts to simplify coding of image processing workflows, and enforces generic programming by using templates. It supports an automated wrapping process that generates interfaces between C++ and interpreted programming languages such as Tcl, Java, and Python. ITK is cross-platform, using the CMake build environment to manage the configuration process.

Parallel image processing on ITK is implemented by the manipulation of multiple threads at a high abstraction level. This approach masks the complexity of thread manipulation on the several platforms supported by ITK. However, until version 3.2, ITK had no official support for GPGPU filter implementations. This was included on version 4 released in Dec.2011, and the reimplementation of filters to the GPU should provide considerable performance gains to ITK's end users, since many applications run on desktop computers.

CUDAITK [11] is a project that tried to implement an abstraction layer allowing the use of CUDA-based filters on ITK. On CUDAITK, CUDA-enabled versions of filters are implemented along with their default CPU counterparts, and the choice over which version to use is controlled by an environment variable at run time. Its architecture requires one data copy from host to device before and another from device to host after the execution of each CUDA filter, and does not allow data to be kept on the GPU. Thus, even when two consecutive filters run on a GPU (on a many-steps composed filter like Canny, for example), data has to be copied to/from GPU/CPU after each step. Since data transfers are extremely time consuming, CUDAITK is of little practical use.

A different approach is implemented by the Cuda Insight Toolkit (CITK) [12], [13]. It is an open source project and was considered on the outline of official GPGPU support for ITK's version 4. CITK consists basically of a modified version of the data container class *ImportImageContainer*, which is a core component of the *itk::Image* class. The new *CudaImportImageContainer* class allows for reasonable compatibility with existing ITK components. It is used to handle image data in both GPU and CPU memories. When some filter requires the data of an image, the class checks where the most recent data related to that image is stored, and returns the appropriate pointer, performing data transfers between host and device only if needed, i.e., when the filter requesting the data is running on a different hardware than the stored image. This way CITK minimizes data copies between CPU and GPU, allowing filter pipelining and mixed use filters running on GPU and CPU in the same pipeline.

Nonetheless, even with an appropriate CUDA programming framework, the implementation of CUDA-enabled ITK workflows must consider the trade off between memory transfer time and processing speedup.

## III. CUDA-based Canny Algorithm for ITK

Implementation of a composed ITK image processing filter is important to identify the main difficulties involved in the integration of CUDA filters to ITK workflows. The Canny Edge Detection was chosen, even though it has already been implemented in other contexts [5], [6], for three main reasons: 1) there is no ITK implementation, and ITK has to get GPGPU support; 2) it contains both trivially parallel and intrinsically sequential parts; and 3) its computation is complex enough so that the speedup surpasses the data transfer time.

The Canny edge detection algorithm begins with a Gaussian filtering to smooth the input image and reduce false edges detection. It then computes the image's gradient magnitude and direction through first or second order derivative operators. First order derivatives like Sobel filter are simpler and faster, but second order derivative operators present better results with a better signal-to-noise ratio and sub-pixel resolution [14]. One commonly used second order derivative computation is based on differential geometry [15], [16]. The next step is called Non-Maximum Suppression (NMS) and consists of setting all pixels that are not maximum at the gradient's direction on a neighborhood to zero. Remaining pixels are subjected to a double threshold hysteresis process to define the final edge pixels.

The specific implementation in this work (CudaCanny) is composed of four CUDA-enabled ITK filters, connected as depicted in Fig.1 and with following steps:

1: Copy input image to GPU global memory (done by CITK);
2: Convolve separable Gaussian kernels on input image resulting in smoothed image $L$;
3: Compute the second and third-order derivative of the smoothed image ($L_{vv}, L_{vvv}$), and the gradient magnitude $L_v$;
4: Detect Zero-Crossings on $L_{vv}$ and multiply the result of Zero-Crossing with gradient magnitude;
5: Use double threshold and hysteresis to define edge pixels;
6: Copy edge image back to CPU memory (done by CITK).

### A. Gaussian Smoothing

Gaussian smoothing is implemented in a similar way as proposed by [5], but instead of having a fixed sized Gaussian kernel, it receives the variance ($\sigma$) as input parameter and computes the two separable Gaussian kernels accordingly.

The next step computes the magnitude and direction of the image gradient. This can be done through first or second order derivative operators. Most CUDA-based implementations of Canny use the Sobel first order derivative [5], [6], while the ITK library implements a second order derivative based on differential geometry [15], [16]. Therefore both versions were

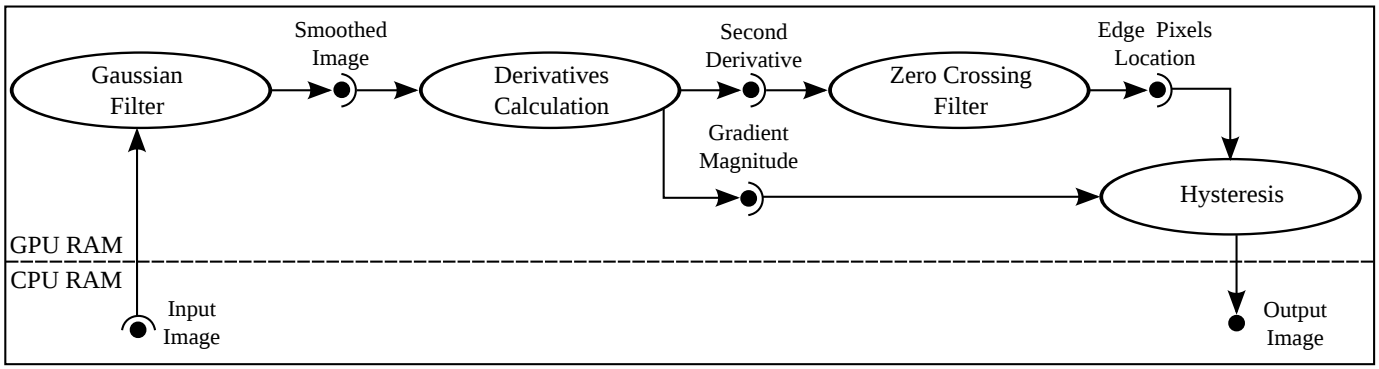



Fig. 1. CudaCanny Filter Pipeline

implemented, although only the second order derivative was benchmarked against ITK standard implementation.

### B. First Order Derivative Sobel Filter

The Sobel filter uses two $3 \times 3$ masks to compute the horizontal and vertical gradient $(G_x, G_y)$ [17]. Gradient magnitude is given by $|G| = \sqrt{G_x^2 + G_y^2}$, whilst the gradient direction is calculated as $\theta = \arctan(G_y/G_x)$ with $\theta \in \{-\pi/2, \pi/2\}$ (Fig.2a). Separable filters can also be used [5], but despite performing less arithmetic operations, separable Sobel filters demand considerably more (slower) memory access operations than non-separable filters with magnitude and direction computed in the same CUDA kernel.

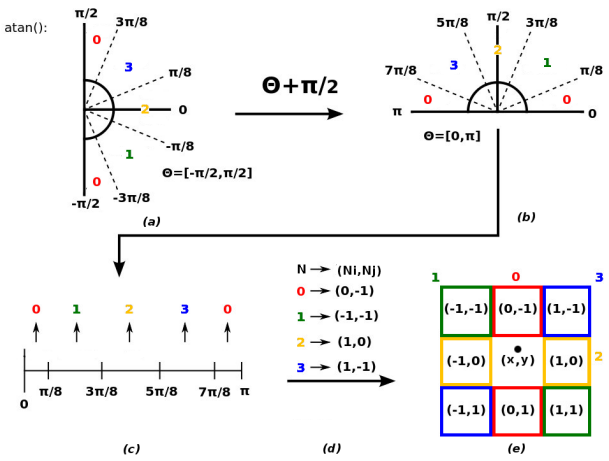



Fig. 2. Computation of the coordinates of the neighbor pixel in gradient direction for the Sobel filter.

Inside the Canny filter, the gradient direction $\theta$ is used to determine the coordinates of the two neighbor pixels in the direction of the maximum gradient (Fig.2e). Therefore, our Sobel implementation also computes the coordinates of these neighbors in a smart way, using $\theta$ as an index, and avoiding conditional expressions and consequent warp serializations.

Given a pixel at coordinates $[i, j]$, and its corresponding $\theta[i, j]$, the coordinates of the neighbor pixels in the direction of the maximum gradient $([i_1, j_1], [i_2, j_2])$ can be efficiently computed in CUDA as shown on the following code extract:

```
theta = θ[i,j] + π/2;   // (Fig.2b)
if (theta > 7*π/8)
   theta -= 7*π/8;
N = (int) ceilf(4*theta/π - 0.5);   // (Fig.2c)
N_i = 1 - (N == 0) + ((N == 1) << 1);   // (Fig.2d)
N_j = (N == 2) - 1;
i_1 = i + N_i;
j_1 = j + N_j;
i_2 = i - N_i;
j_2 = j - N_j;
```

### C. Second Order Derivative Gradient

The implemented second-order derivative operator produces edges with a better signal-to-noise ratio and sub-pixel resolution [14] than the Sobel operator. It is based on differential geometry [15], [16] and introduces, at every image point, a local coordinate system $(u, v)$, with the $v$-direction parallel to the gradient direction on the smoothed image $(L)$. The CUDA kernel to calculate the second derivative on the $v$-direction $(L_{vv})$, works as follows:

1: Each thread is associated to one pixel from the smoothed image $L$;
2: Texture cache is used to obtain the pixel and its 8 neighbor values from global memory;
3: Compute the local partial derivatives $L_x$,$L_y$,$L_{xx}$,$L_{xy}$, $L_{yy}$ and the second-order directional derivative in the $v$-direction $L_{vv}$ according to [15]

Another CUDA kernel is used to calculate the gradient magnitude (first derivative on the $v$-direction) $L_v$, as follows:

1: Each thread is associated to one pixel from the smoothed image $L$ and second-order derivative $L_{vv}$;
2: Texture cache is used to obtain the pixel from both images and its 4 neighbor values from global memory;
3: Compute the local partial derivatives $L_x$,$L_y$,$L_{vvx}$, $L_{vvy}$ and the third-order directional derivative in the $v$-direction $L_{vvv}$ according to [15];

4: Compute the gradient magnitude:

$$L_v \leftarrow \begin{cases} \sqrt{L_x^2 + L_y^2}, \textbf{ if } L_{vvv} \leq 0, \\ 0, \textbf{ otherwise}. \end{cases}$$

### D. Non-maximum Suppression

Non-maximum Suppression is performed by finding the zero-crossing pixels on the second-order derivative ($L_{vv}$). Zero-crossing occurs at positions where signal changes or null values followed by nonzero values appear in a pixels 4-neighborhood. The CUDA kernel to compute zero-crossing works as follows:

1: Each thread is associated to one pixel of the second-order derivative $L_{vv}[i,j]$;
2: Texture cache is used to obtain the pixel ($p = L_{vv}[i,j]$) and its 4 neighbor values ($left, up, right, down$) from global memory:
3: Test for zero-crossing ($zC[i,j]$) on the 4-neighborhood:

$$zC[i,j] \leftarrow \begin{cases} 1, \textbf{if } \text{zero-crossing occurred}, \\ 0, \textbf{otherwise}; \end{cases}$$

Verification of zero-crossing between the central pixel and its 4-neighborhood (Step 3:) can be implemented without using conditional expressions as shown on following code excerpt:

$$\begin{array}{l} zC[i,j] = (p * left <= 0) * (|p| < |left|); \\ zC[i,j] = zC[i,j] \;\; || \;\; (p * up <= 0) * (|p| < |up|); \\ zC[i,j] = zC[i,j] \;\; || \;\; (p * right <= 0) * (|p| < |right|); \\ zC[i,j] = zC[i,j] \;\; || \;\; (p * down <= 0) * (|p| < |down|); \end{array}$$

.

### E. Hysteresis Thresholding

The last step of the Canny algorithm is the hysteresis thresholding, implemented in two CUDA kernels. The first kernel performs a double threshold:

1: Each thread is associated to one pixel of the zero-crossing ($zC[i,j]$) and gradient magnitude ($L_v[i,j]$);
2: $edge[i,j] \leftarrow L_v[i,j] \times zC[i,j]$;
3: Perform a double thresholding to classify pixels as Definitive Edges (DE), Possible Edges (PE) or Non-Edges (NE):

$$edge[i,j] \leftarrow \begin{cases} \text{DE, } \textbf{if } edge[i,j] > T_H, \\ \text{PE, } \textbf{if } T_L < edge[i,j] \leq T_H, \\ \text{NE, } \textbf{otherwise}; \end{cases}$$

In order to avoid warp serialization, the double thresholding on step 3 can be implemented as follows:

```
#define DE 255
#define PE 128
#define NE   0
edge[i,j] = ((PE-1) * (edge[i,j] > T_H)+PE)*
                (edge[i,j] > T_L);
```

.

After double thresholding, $PE$ pixels might become $DE$ if they have any $DE$ in a 4-neighborhood. The traditional strategy for hysteresis uses a $DE$ queue, and for each pixel on the queue, follows along its $PE$ 4-neighborhood changing them to $DE$ and putting them into the queue, on an interactive process that ends when the queue is empty.

This strategy is not efficient on GPGPUs because of its high data interdependency, and the possibility of a thread having to access pixels all over the image, completely messing up memory access coalescence. The proposed algorithm therefore departs from classical CPU implementations, and closely follows the algorithm proposed in [5]. A synchronization algorithm runs on CPU and has a control variable (modified_global) that is changed whenever a kernel changes the status of a pixel:

1: **repeat**
2: modified_global ← **false**;
3: Copy modified_global from CPU to GPU;
4: Run hysteresis edge following CUDA kernel
5: Copy modified_global from GPU to CPU;
6: **until** modified_global = **false**;
7: **for all** Remaining pixels labeled $PE$ **do**
8: label them as $NE$;
9: **end for**

The hysteresis edge following kernel uses a 324 position array to store an image region of size $18 \times 18$ on shared memory. The $16 \times 16$ center pixels of the region are then processed by a 256 threads block. The extra one-pixel width border overlaps neighbor regions of the image. Despite dividing the image in blocks of size $16 \times 16$, the algorithm supports images of any size, which is not the case on [5]. Two shared variables (modified on global memory and modified_region on shared memory) controll the algorithm loops:

1: Each thread block loads a region of $18 \times 18$ pixels into shared memory;
2: Each thread is assigned to a pixel of the $16 \times 16$ inner pixels of the region;
3: **repeat**
4: modified_region ← **false**;
5: Synchronize all threads in the same thread block;
6: **if** pixel = $PE$ **and** any pixel in 4-neighborhood = $DE$ **then**
7: pixel ← $DE$;
8: modified_region ← **true**;
9: **end if**
10: Synchronize all threads in the same thread block;
11: **if** modified_region = **true then**
12: modified_global ← **true**;
13: **end if**
14: **until** modified_region = **false**;

## IV. Methodology

### A. Hardware

Two computers (CPUs) and three NVidia GPGPUs with following hardware configuration were used on the tests:

- **C2D**: Intel Core2Duo, two 2.8-GHz cores, 3072KB cache and 2GB RAM;
- **Ci7**: Intel Core i7-975, four 3.33-GHz cores, 8192KB cache and 12GB RAM;
- **G80**: NVidia GeForce 8800 GT with 112 1.5-GHz cores and 512MB RAM.
- **GT200**: NVidia Tesla C1060 with 240 1.3-GHz cores and 4GB RAM;
- **Fermi**: NVidia Tesla C2050 with 448 1.15-GHz cores and 3GB RAM.

### B. Image Databases

Four databases $(B_1, B_2, B_3, B_4)$ with $100$ images each were used to test the developed algorithms. The $B_1$ database consists of the Berkeley Segmentation Dataset [18], [19]. The additional $B_i$, $i \in \{2, 3, 4\}$ databases were created by replicating each image of the $B_{(i-1)}$ database vertically and horizontally (tilling), composing a new image that has 4 times more pixels that the originating image. For example: the $B_2$ image database was created by replicating each $B_1$ image. Image sizes for each image database are: $B_1 = 321 \times 481$ pixels, $B_2 = 642 \times 962$ pixels, $B_3 = 1284 \times 1924$ pixels and $B_4 = 2568 \times 3848$ pixels.

(a) $B_1$ image (b) Replicated $B_2$ image

Fig. 3. Replication example used to create bigger images

Image sizes were chosen so that they always fit into the GPU memory. No blocking strategy was implemented to deal with larger images or regions of interest, as defined by ITK. Image pixel type were always single precision floating point (**float**), and type conversions were not considered in time measurements.

It is also important to notice that we do not intend to compare the accuracy of a new edge detection filter, so that it could be influenced by the image tilling process. Although we have to make sure that the CUDA implementation is equivalent to the ITK implementation for a reasonable set of images, the content of these images is almost irrelevant, as long as they present enough edge variations for the Canny algorithm.

## V. Experiments and Results

### A. Conformance Test

The Conformance test aims to evaluate the similarity between edges detected by CudaCanny and the original ITK Canny implementation (ItkCanny). Edges generated by ItkCanny are considered the reference, and the CudaCanny must return an equivalent set of edges. In order to verify similarity of the edges, three quality metrics proposed in [20] were used:

- $P_{co}$: percentage of edge pixels detected by both implementations;
- $P_{nd}$: percentage of edge pixels detected by ItkCanny that were not detected by CudaCanny (false negatives); and
- $P_{fa}$: percentage of edge pixels detected by CudaCanny that were not detected by ItkCanny (false positives).

For all databases the correctly detected edge pixel percentage ($P_{co}$) was higher than $99.5\%$, and less than $0.5\%$ of all edge pixels were detected by only one of the detectors ($P_{nd}$ and $P_{fa}$). This leads to the conclusion that the implemented CudaCanny ITK filter can be used as a replacement for the CPU-based itkCanny filter on any ITK image processing workflow.

Another interesting aspect is that larger image databases have lower error rates ($P_{nd}$ and $P_{fa}$). This happens because most edge detection discrepancies occur at the image borders. Since larger images have a greater inner pixels to border pixels ratio, the error rate decreases in the same proportion.

### B. Performance Tests

The Performance Tests aim to evaluate the execution time of ItkCanny and CudaCanny algorithms on the described CPUs (test IDs: *ITK-C2D* and *ITK-Ci7*) and NVidia GPGPUs (test IDs: *G80*, *GT200* and *Fermi*), respectively. Execution times were measured considering the elapsed time of the *Update()* method of the ITK classes, which is invoked by ITK work-flow manager upon filter execution. Partial times were recorded for each step of the Canny algorithm, and on the CudaCanny implementation memory transfer to/from GPU is also considered (I/O time).

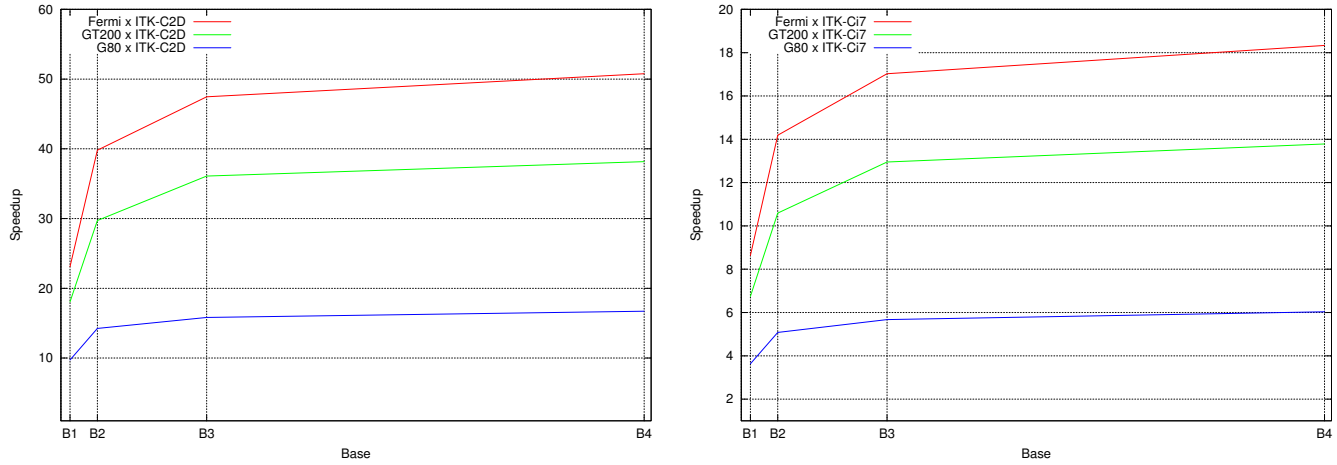
The sum of the execution times of the Canny filter for all images in an image database is the image database execution time. Each edge detector, on each hardware, was averaged over $100$ executions for each image database.

OpenCV[2] Canny implementation (cvCanny) was also evaluated. In this case, no direct comparison is possible, since cvCanny uses the Sobel for gradient computation, but it makes extensive use of CPU multithreading and Streaming SIMD Extensions (SSE), being a good reference implementation for CPU. These tests are referred to as *CV-C2D* and *CV-Ci7*.

Both Conformance and Performance tests used a Gaussian variance $\sigma = 1.4$, higher hysteresis threshold $T_H = 7$ and lower threshold $T_L = 4$.

On Fig.4 the good performance of CudaCanny over ItkCanny is highlighted. Speedup varies from $10$ to $50$ times when compared to the ItkCanny C2D, and $4$ to $18$ times when compared to the Ci7. The figure reveals that CudaCanny outperforms ItkCanny on all tested hardware by almost two orders of magnitude. This is far too much performance gain to be attributed only to the hardware, and is due to a more efficient implementation of CudaCanny (worth to notice that ITK does not claim to be efficiency driven, which it really is not).

[2]http://opencv.willowgarage.com/wiki/




(a) GPU x ItkCanny C2D

(b) GPU x ItkCanny CI7

Fig. 4. Speedup of CudaCanny (G80, GT200, Fermi) compared to the ItkC2D (a) and ItkCi7 (b).

The distance between the labels on the X coordinates of Fig.4 is proportional to the number of pixels of the images in the corresponding database. It is possible to observe that, as the image size increases, the speedup in the less parallel part of the program (I/O and hysteresis) are responsible for the smaller speedup.

Execution times for ItkCanny, cvCanny and CudaCanny on each hardware and for each image databases are presented in Fig.5. Graph bars are in logarithmic scale to allow better comparison.

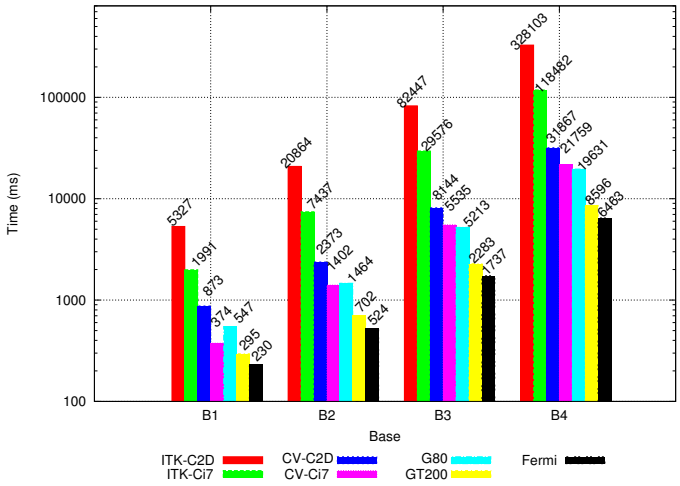



Fig. 5. Execution time (log scale) of the ItkCanny, cvCanny and CudaCanny algorithms for each database and hardware.

As can be seen, CudaCanny slightly outperforms cvCanny in most cases. The cvCanny execution times show it is much more efficient than ItkCanny, and this cannot be attributed to the use of a different gradient computation method, since this is a very little time consuming step. The reason is that cvCanny makes a much better user of thread parallelization and SSE instructions, besides having a lighter programming structure than ITK. The execution time differences of cvCanny and CudaCanny are similar to those reported on [5], although there the I/O time was not considered.

Fig.5 also shows that the CudaCanny execution was $\approx 3.0$ times faster on Fermi than on G80. This is a very reasonable scaling performance, since I/O time is almost the same and the Fermi GPU has 4 times more processing cores and a L2 cache to global memory.

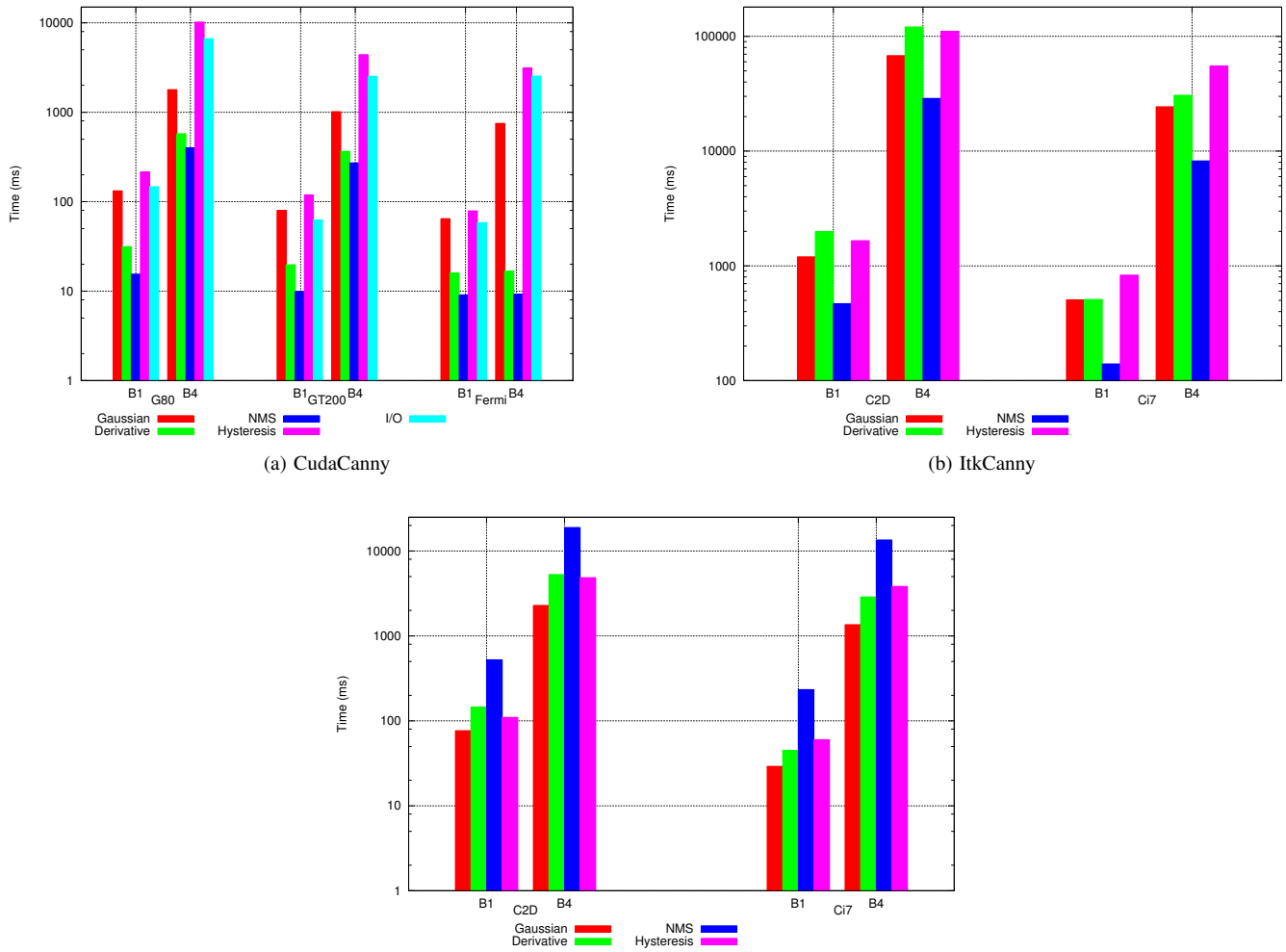
Comparing to the GT200 GPU, Fermi was only $1.27$ to $1.33$ times faster, despite having almost twice as much processors. This can be explained on one hand by the fact that the GT200 already implements almost the same global memory access rules, so that less improvement is made on concurrent memory access. On the other hand, as shown on Fig.6(a), most of the execution time is spent on I/O and Histeresys, which are not much influenced by the architecture changes.

Fig.6(a), 6(b) and 6(c) present the execution time of each routine of the CudaCanny, ItkCanny and CvCanny, respectively, for databases $B_1$ and $B_4$.

On CudaCanny, I/O and hysteresis time do not scale as processor numbers or image sizes increase. The Gaussian kernel scales a little better, but the Derivative and NMS scale really well, because they do not depend on much global memory access and have almost no warp divergence

CudaCanny spends more time on the Hysteresis procedure than on the Non-Maximum Suppression, whilst on CvCanny it is the other way around. This is due to different implementations of the routines, since on CvCanny part of the Hysteresis is done at the NMS routine. Therefore, in order to compare CudaCanny and CvCanny, it is more appropriate to sum the times spent on both routines.

As can be observed, CudaCanny significantly outperforms CvCanny on these routines ($\approx 10.6s$ on G80 $\approx 4.6s$ on GT200 and $\approx 3.1s$ on Fermi, versus $\approx 23.7s$ on a C2D and $\approx 17.3s$ on a Ci7, for base $B4$), due to a parallel implementation that does not use a FIFO. Instead of push/pull candidate edge




(a) CudaCanny

(b) ItkCanny

(c) CvCanny

Fig. 6. Execution time of (a) CudaCanny, (b) ItkCanny and (c) CvCanny on each part of the program, for databases $B_1$ and $B_4$.

pixels to/from a FIFO, a strategy that splits the image among the processing cores, and repeatedly iterates until no changes happen, allows parallel execution without blocking.

## VI. Conclusion

This paper presented a CUDA based implementation of the Canny edge detection filter for the ITK library. Considering a typical desktop setup for ITK applications (C2D), the use of a simple GPGPU card (G80) can provide speedup of almost 20 times. CudaCanny outperformed the standard CPU implementation on all image databases and graphic cards. Even when compared to a top line Core i7 processor, GPUs obtained 5 to 18 times speedups.

Two major contributions of this paper were: 1) the development of an efficient CUDA-based Sobel algorithm to compute edge magnitude and direction, avoiding conditional expressions and reducing memory access; and 2) the implementation of an efficient Canny edge detection algorithm, using second order derivatives, and a hybrid CPU-GPU approach for the final hysteresis thresholding step.

The development of image processing filters for GPGPUs can be considerably optimized if the programmer can avoid warp serialization, as shown on many of the implemented filters. Frequently used algorithms are special candidates for cutting edge reimplementation.

## Acknowledgment

The authors would like to thank CNPq for financial support of the first author.

## References

[1] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, 1986.

[2] S. S. Stone, H. Yi, J. P. Haldar, W. mei W. Hwu, B. P. Sutton, and Z. pei Liang, "How gpus can improve the quality of magnetic resonance imaging," in *In The First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[3] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 15–25.

[4] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision," in *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*. New York, NY, USA: ACM, 2005, pp. 849–852.

[5] Y. Luo and R. Duraiswami, "Canny edge detection on nvidia cuda," *Computer Vision and Pattern Recognition Workshop*, vol. 0, pp. 1–8, 2008.

[6] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Parallelization of a video segmentation algorithm on cuda-enabled graphics processing units," in *Euro-Par '09*. Berlin: Springer-Verlag, 2009, pp. 924–935.

[7] R. Palomar, J. M. Palomares, J. M. Castillo, J. Olivares, and J. Gómez-Luna, "Parallelizing and optimizing lip-canny using nvidia cuda," ser. IEA/AIE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 389–398.

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[9] V. Podlozhnyuk, "Image convolution with cuda," June 2007. [Online]. Available: http://developer.download.nvidia.com

[10] "Insight segmentation and registration toolkit," 1999. [Online]. Available: http://www.itk.org

[11] W.-K. Jeong, "Cudaitk," 2007. [Online]. Available: http://www.cs.utah.edu/~wkjeong/

[12] R. Beare, M. Kuiper, D. Micevski, C. Share, L. Parkinson, and P. Ward, "Cuda insight toolkit," 2010. [Online]. Available: http://code.google.com/p/cuda-insight-toolkit/

[13] R. Beare, D. Micevski, C. Share, L. Parkinson, P. Ward, W. Goscinski, and M. Kuiper, "Citk - an architecture and examples of cuda enabled itk filters," *The Insight Journal*, 08 2011.

[14] J. Canny, "Finding edges and lines in images," MIT, Tech. Rep., 1983.

[15] T. Lindeberg, "Edge detection and ridge detection with automatic scale selection," *Int. J. of Computer Vision*, vol. 30, 1998.

[16] H. K. Kidwai, F. N. Sibai, and T. F. Rabie, "Parallelization and performance evaluation of an edge detection algorithm on a streaming multi-core engine," *JITR*, vol. 2, no. 4, pp. 81–91, 2009.

[17] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[18] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proc. 8th ICCV*, vol. 2, July 2001, pp. 416–423.

[19] P. Arbelaez, C. Fowlkes, and D. Martin, "The berkeley segmentation dataset and benchmark," 2007. [Online]. Available: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

[20] I. A. G. Boaventura and A. Gonzaga, "Método de avaliação de detector de bordas em imagens digitais," *Anais do V Worskhop de Visão Computacional*, 2009.