Fuzzy Logic-Based Image Processing Using Graphics Processor Units

R. H. Luke, D. T. Anderson, J. M. Keller
Department of Computer Engineering
University of Missouri
Columbia, Missouri, USA
{rhl3db, dtaxtd}@mizzou.edu, kellerj@missouri.edu

S. Coupland
Centre for Computational Intelligence
De Montfort University
Leicester, LE1 9BH, UK
simonc@dmu.ac.uk

Abstract—**This paper introduces a parallelization of fuzzy logic-based image processing using Graphics Processor Units (GPUs). Using an NVIDIA 8800 Ultra, a 126 time speed improvement can be made to fuzzy edge extraction making its processing real-time. The GPU can process approximately 42 frames per second at 640x480 image resolution, thus 307,200 inference processes per frame. With a computational speed improvement of over two orders of magnitude, more time can be allocated to higher level computer vision algorithms. This GPU solution is described using NVIDIAs Compute Unified Device Architecture (CUDA).**

Keywords- ***Graphics Processor Units, Fuzzy Image Processing, Fuzzy Edge Extraction, Compute Unified Device Architecture***

I. INTRODUCTION

Edge extraction is a low level technique used in image processing and computer vision. There are many algorithms to perform edge extraction with a wide range of computational complexities and performance. Common approaches include the Sobel [1] and Prewitt [2] convolution kernels and the Canny Edge Detector [3]. Edges are most often used as low level information fed to a higher level process such as segmentation or recognition.

Some more recent algorithms have merged the strengths of image processing and fuzzy logic. These algorithms span a wide range of applications such as image enhancement [4], edge extraction [5], segmentation [6], and contrast adjustment [7]. An edge detection model based on Fuzzy Logic, called Fuzzy Inference Ruled by Else-Action (FIRE), was designed by Russo and Ramponi in [8]. This algorithm checks eight unique edge cases and outputs a value related to the confidence of the best matched case. The significance of this specific edge detection algorithm is its robustness to noise.

Even with simple algorithms, there is a large amount of work required to perform edge detection on moderately sized images. The workload is further compounded for real-time systems processing multiple frames per-second. A stream processing solution based on Graphics Processor Units (GPUs) and NVIDIAs Compute Unified Device Architecture (CUDA) is presented. While the work in this paper focuses on edge detection as the parallel execution of a single Fuzzy Inference System (FIS) for multiple different inputs, the GPU solution can be easily modified to fit a wide range of batch-processing tasks that utilize the standard Mandani-type FIS [9][10][11]. The system's extension is discussed in a later section.

It should be noted that we could have implemented any fuzzy logic-based image processing technique. We chose to demonstrate the proposed GPU-based architecture using FIRE because it is an easily understood algorithm. Our goal is not to show the performance of FIRE, but rather to present a framework for the parallel execution of fuzzy logic based image processing.

Several hardware accelerated FIS solutions have been developed to improve processing speed. For example, FPGA [12][13], and VLSI [14] solutions exist, however the significant portion of these are related to Fuzzy Control, not Image Processing. These specialized solutions are generally expensive and time consuming to modify and adapt to a new problem. In contrast, GPUs are relatively inexpensive, easily integrated with PCs and flexible to general purpose programming.

Using GPUs for general purpose computing is not a new idea. Well-known graphics API-based approaches include: Fast Fourier Transform (FFT) [15], linear algebra operators [16], protein folding [17], Fuzzy C-Means [18][19], and the execution of a single FIS [20]. Researchers have also begun to recently explore CUDA in Image Processing, such as: optical flow estimation [21], segmentation of medical images [22], and the Canny Edge Detector [23].

II. EDGE EXTRACTION USING FIRE

An input image I has luminance values from $\{0,1,\ldots,255\}$. For pixel P_{ij}, let $W_{ij} = \{P_{(i-e)(j-f)},\ldots,P_{(i+e)(j+f)}\}$, where $e,f \in \lfloor 0,\ldots,K/2 \rfloor$ and K is the window size, be the set of pixels in the neighborhood of P_{ij}, with the exception of P_{ij} (hence, $|W_{ij}| = K*K-1$). The inputs to the fuzzy edge extractor are the luminance differences between pixels in W_{ij} to P_{ij},

$$X_{(ij,m)} = W_{(ij,m)} - P_{ij}, \quad (1)$$

$$1 \le m \le (K*K-1),$$

ISBN: 978-989-95079-6-8

where $W_{(ij,m)}$ is the m[th] element in the window centered at P_{ij}. All subsequent operations performed on $X_{(ij,m)}$ are independent of all other pixels, therefore, $X_{(ij,m)}$ will be notationally reduced to X_m, to simplify the indexing and as all discussion is about a specific P_{ij} and its W_{ij}. The domain of X_m is the set $\{-255,\ldots,255\}$.

As stated previously, FIRE is a case-based system used to test eight unique edge orientations on a window. The size of the window used in this system is 3x3. Therefore, X has a cardinality, $|X|$, of 8.

An overly simplified rule is: *If a pixel lies upon an edge Then make it black Else make it white.* Eight unique cases, representing an edge at every 45 degrees of rotation, figure 1, is represented as a rule. Each rule has six antecedents, coinciding with six of the eight pixels in X. Fuzzy sets are used to represent the linguistic terms negative and positive. Similarly, the linguistic uncertainties of the consequent terms white and black are also modeled as fuzzy sets.

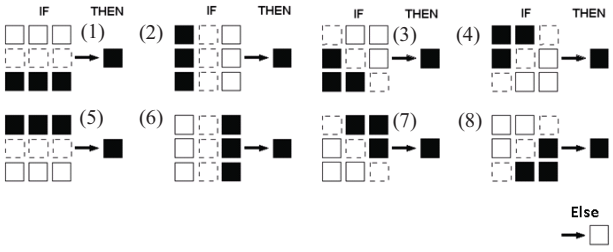

Figure 1. A graphical representation of the eight rules in the fuzzy rule base. Each rule has six antecedents represented as negative (black boxes) and positive (white boxes) luminance differences to the center pixel. The consequent is the same for each rule, a black pixel. A single else case handles the situation of none of the antecedents being matched.

For each pixel, each of the eight rules consisting of six antecedents are fired using the values in X. For each rule, the average of the six confidences is used as the total antecedent firing, f_i, where $1 \le i \le 8$. In this algorithm, it is assumed that a given pixel can belong to, at most, one of the eight cases. Therefore, the rule with the largest firing f_l is used as the antecedent firing strength to the fuzzy consequent set representing black.

$$f_l = \max(f_1,\ldots,f_8) \quad (2)$$

This is a departure from the more common Mamdani FIS which uses all rules in its decision making. The implication of the white fuzzy set uses $f_s = 1 - f_l$, i.e. the complement of the antecedent. The centroid of the aggregation of implication using f_l and f_s is the output of the algorithm. It should be noted that the result of the algorithm, $\{P'_{11},\ldots,P'_{ij},\ldots\}$, is in [0, 127.5], not [0, 255]. This is due to the setup of the antecedent fuzzy sets. In the extreme case, if all pixels in the 3x3 window are the same intensity, (no edge), all antecedent sets for all rules are fired with a 0.5 membership. The output of equation 2 will be 0.5 and therefore, f_s will also be 0.5. The consequents will have the same mass, and the final result will be 127.5. The domain of P'_{ij} is discretized and its range expanded to $\{0,1,\ldots,255\}$ by multiplying a constant, C, by P'_{ij}. In this paper, $C = 2$. Figure 2 displays an input and output image using this system.

Figure 2: Example output image of the algorithm described in [8].

III. CUDA INTRODUCTION

Because the calculation of the edge confidence at each pixel is independent of all other pixel calculations, this algorithm is a good candidate for parallelization. A single FIS is run for each pixel in an image. For a 640x480 image this results in the execution of a single FIS for each of the 307,200 pixels. Parallelization and implementation of the FIRE system to take advantage of the large number of co-processors on a GPU greatly increases the speed over sequentially processing the elements on the CPU.

NVIDIA defines CUDA as "a hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device" [24]. The GPU is a highly parallel general purpose co-processor, not just a graphics processor. The CUDA API extends the C language and allows the largest portion of the programming community a quick transition to its use.

CUDA allows multiple programs, kernels, to run sequentially on a single GPU. CUDA organizes a kernel into a grid. A grid is subdivided into blocks. All blocks run the same kernel, but each runs independently from all others. Each block is made up of threads, the smallest divisible unit on the GPU. The actual work of the kernel is performed at the thread level.

The hierarchical structure of CUDA provides a mechanism to take advantage of the underlying hardware structure. This is mainly due to the grouping of processors into multiprocessors and that the memory types are unique to the grid, blocks and threads. Each multiprocessor is comprised of Q processors, Q=8 for the NVIDIA 8800, and each block is loaded into a single multiprocessor. Each multiprocessor is a Single Instruction Multiple Data (SIMD) set, meaning that multiple elements, Q, will be processed in parallel using the same instruction but different data. Threads in a block are partitioned into warps, where a warp size in the NVIDIA 8800 is 32. All threads in a warp are executed using the same

ISBN: 978-989-95079-6-8

instruction. Conditional logic can cause warp divergence and result in performance degredation. Code must be designed to take the warp size and control flow into account.

There are several types of memory in CUDA available to the programmer. Using the proper memory is vital to the efficiency of a program. Therefore, developers must create algorithms with memory access in mind.

The Global memory type can be read from and written to by any thread. Global memory access is the slowest of all types and should be used sparingly. The Constant and Texture memories can be read from any thread in a kernel, but not written to. The access times of texture memory are substantially smaller than Global memory.

Shared memory can be read from and written to by any thread associated with the block. This memory is local to the specific block and cannot be accessed by threads of any other block. Memory access times are shorter than the types mentioned above. Values from Global memory are often brought into shared memory at the beginning of a kernel, all threads are synchronized, and future repetitive access to this data is very efficient.

At the thread level, there is Local Memory and Registers which have the smallest access times. These memory locations are only accessible from a single thread. The entire memory layout is shown in figure 3.

IV. CUDA FIRE IMPLEMENTATION

This GPU implementation of the FIRE algorithm consists of two kernels. The first kernel performs all pixel neighborhood differencing, fuzzification for all rules, computes the average of the antecedent firings for each rule, and finds the maximum rule firing. The second kernel uses the minimum operator for implication, a maximum operator for rule aggregation and two sum reductions [25] to determine the centroid of the aggregated consequent sets for defuzzification. Though a simpler type of defuzzification could have been performed, we elected to use the standard general centroid defuzzification so that readers could easily extend this work to other more complex fuzzy-logic problems.

Even though more parallelism could be performed at each step, this two kernel approach is faster for three reasons. First, if the first kernel is subdivided into more parallel stages, each stage does too little computation relative to the amount of memory access. The GPU is most efficient when there is a large arithmetic to memory access ratio. Secondly, the second stage, implication, takes advantage of shared memory, making it more efficient to be a single kernel. Finally, most fuzzy systems use a small number of rules, less than 32, which does not justify further kernel subdivision.

The first kernel divides the input image into 8x8 non-overlapping pixel blocks. Each image block is processed in a CUDA block. This is done to make the processing of pixels faster by loading blocks of pixel data into shared memory for threads in a CUDA block to work on. As described earlier, the algorithm works on 3x3 windows of data for each pixel. In order to perform these operations on the pixels of the boundary of the 8x8 image blocks, a one pixel border must also be brought into memory. So, a 10x10 window of pixels is brought from texture memory into a block's shared.

Each thread can now perform the operations of the algorithm. An array of length eight, local to each thread, is defined and filled with the pixel neighborhood difference values (X_m , i.e. equation 1). The six antecedents of the eight rules are fired using the values in X . The antecedent firings are then summed across each rule. The max of these eight values is found and divided by six to find the average. This value is then written to the output of this kernel.

The output of the first kernel, f_1 , is the average of the maximum fired antecedent for each pixel of the input image and is in [0, 1]. For the second kernel, a separate CUDA block is created for each f_1 . Each of the T threads in a block performs inference aggregation, and is used to find the centroid of the aggregated consequent sets. Each consequent set is discretized into 256 elements. We empirically picked a T of 128, however, the selection of T is related to reduction, which is detailed later.

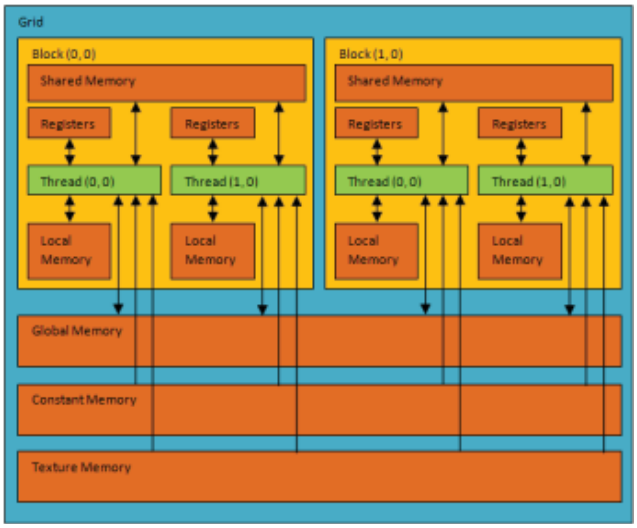

Figure 3: Processing organization scheme and memory layout for the GPU using CUDA. Global, Constant and Texture Memories are accessible by all threads. Shared Memory has is faster, but is local to each Block. Registers and Local Memory are the fastest, but are local to each thread.

Each thread performs inference (min) of f_1 with its corresponding discrete consequent domain membership value in the fuzzy set "black". The minimum of f_s and the appropriate discrete consequent domain membership value in the fuzzy set "white" is calculated, and each thread stores their aggregation (max) value in a shared memory array. Each thread is assigned an identifier in CUDA and this identifier is what is used to select the index into the discrete consequent domain fuzzy sets and the shared memory region. For a discrete consequent domain of size 256, a T of 128 was selected, where T was empirically determined. This means each thread calculates two discrete consequent domain elements instead of one. The reason for doing this is related to reduction, which is used for calculating the centroid.

ISBN: 978-989-95079-6-8

Reduction is the next step and there are many guidelines to follow, [25]. One rule states that it is generally better to calculate a few loads and reduction steps at the beginning versus allocating more threads, which become unused as reduction steps proceed. Other guidelines include sequential addressing and unrolling. Each block calculates the numerator and denominator separately and in parallel. The kernel must now compute the sum of the numerator and denominator. Reduction [25], which is the collapsing of a set of elements by some operator into one scalar, (such as the average, sum or max), is a common GPU operation. This procedure usually takes $\log_2(N)$ time, where N is the number of elements to reduce. For large values of N, such as N=1,000,000, a multiple block reduction would be used. However for 256 elements, a single block method keeping all in shared memory is the most efficient. An example of sum reduction in shown in figure 4.

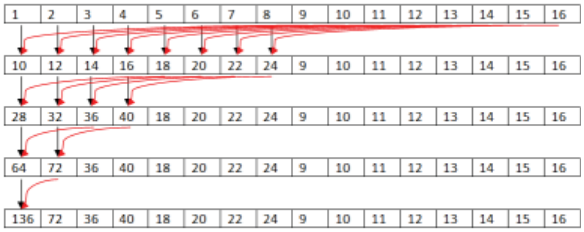

Figure 4: An example of sum reduction. In the first row, eight threads sample two locations in memory, add their values and output the result to a specified location. In the next row, four threads perform the operations. Then, two and one in the final two rows. The total sum is in the first index of the array in the bottom row.

It should be noted that GPU reduction is faster than a single CPU, however, not all processing cores can be used efficiently. This is because an increasing number of processors go unused with each reduction step. For example, reducing 256 elements with 128 threads will reduce to 128 elements in the second step. However, in the second step, only 64 of the threads are involved in reduction. In the third step only 32 are performing work, 16 in the next, and so on.

The final operation of the kernel is the calculation of the centroid. As previously discussed, this value is in [0, 127.5] and is multiplied by 2 (C). It is then discretized so that the final values of the output image, P'_{ij}, are in $\{0,1,\ldots,255\}$. The entire GPU solution is displayed in figure 5.

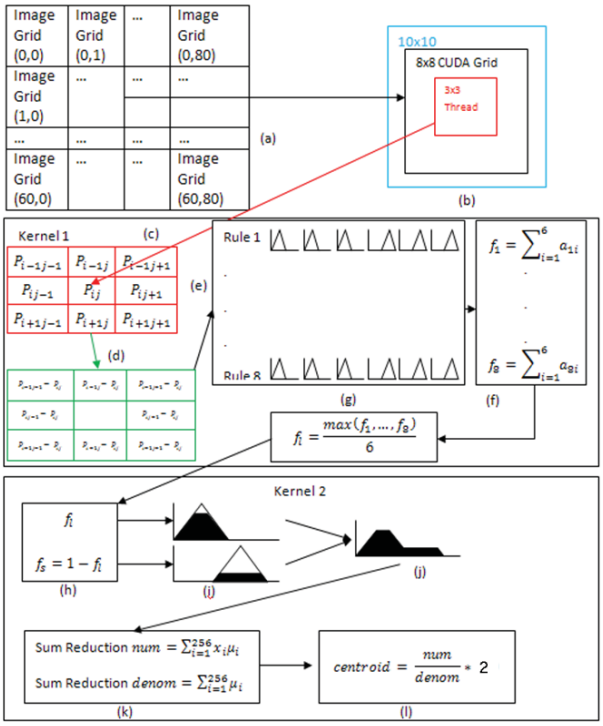

Figure 5: A graphical representation of the system defined in this paper. (a) The 640x480 input image divided into 8x8 blocks. (b) A single CUDA block with a 10x10 shared memory. (c) 3x3 input to a thread. (d) The difference values computed from (c). (e) Antecedent firing over eight rules. (f) Find the average of each rule. (g) Output the largest antecedent firing for kernel 1. (h) Compute else firing. (i) Perform implication on the consequent sets. (j) Aggregate consequent sets. (k) Sum reduction of the numerator and denominator values for centroid computation. (l) Compute the centroid of the aggregated consequents and scale to image output.

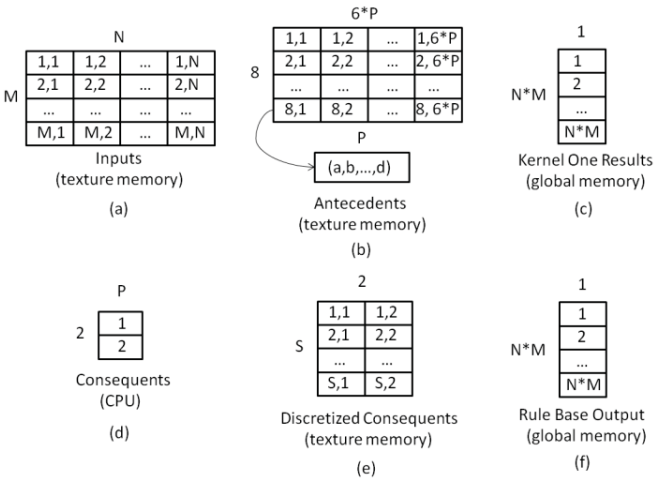

Figure 6: CPU memory and GPU global and texture memory organization for the GPU FIRE solution. For this paper, M = 640, N = 480 and S = 256, however, M, N and S can take on different profiles. (a) The input image in texture memory, (b) antecedents in texture memory, (c) global memory is used to store the output of the first kernel, (d) CPU side parameters of the two consequent sets, (e) GPU texture memory for the discretized consequent sets, and (f) global memory is used to store the output of the second kernel (image of edge confidences).

ISBN: 978-989-95079-6-8

As mentioned above, memory usage and organization is vital to CUDA efficiency. The input image is moved from the CPU to GPU texture memory. In this case, the image size is 640x480. Texture memory is used instead of the slower global memory because the image is read-only on the GPU. Secondly, the antecedent sets for the rule base are stored in texture memory. There are 8 rules with 6 antecedents per rule, and each antecedent set is made up of 4 ordered points, {a,b,c,d} for a trapezoid. Because the rules are read-only, texture memory is used. The output of the first kernel is stored in a one dimensional global memory segment. Next, the discretized consequent sets are transferred from the CPU memory to GPU texture memory. Finally, the output of kernel 2 is stored in a one dimensional global memory segment. The memory organization is displayed in figure 6.

RESULTS

The CPU used for testing was an AMD Athlon 64 FX-55 running at 2.6 GHz. The system had 2GB of RAM and the operating environment was Windows XP. The GPU was an NVIDIA 8800 BFG Ultra with 768 MB of memory [26]. This GPU has 128 processors and connects to the motherboard using PCI express 16X .

The CPU and GPU implementations were then run over a series of different image sizes. Table 1 shows the setups and the associated timings

TABLE I. PROCESSING TIMES AND SPEED INCREASE FROM CPU TO GPU

	320x240 Image 76,800 Pixels	640x480 Image 307,200 Pixels	960x720 Image 691,200 Pixels	1280x960 Image 1,228,800 Pixels	1280x1024 Image 1,310,720 Pixels
CPU	0.73	2.86	6.6	11.5	12.3
GPU	0.006	0.024	0.052	0.091	0.098
Speed Increase	121X	119X	126X	126X	125X

Figure 7: Images processed using the algorithm defined in this paper.

Two output images are shown in figure 7. These images are consistent with the output images from the original paper [8].

As it relates to global memory access, it was found that in the second kernel, the block format and global memory access pattern (index calculation per block) has a large performance impact. For example, two possible block formats for a 640x480 image are (38400, 8) and (8, 38400). A global memory read index (sampling of the f_l value) would then be $b_x * 8 + b_y$ for (38400, 8) and $b_x * 38400 + b_y$ for (8, 38400), where b_x and b_y are the block x and y indices respectively. It was noticed that the (38400, 8) format was three times faster than the (8, 38400) format. In fact, for all profiles (N_{b_x}, N_{b_y}), where N_{b_x} and N_{b_y} are the number of blocks in the x and y dimensions of the grid, it holds that greater speeds are produced if N_{b_y} is a small power of 2 (hence 8, 16, 32), resulting in a smaller calculated offset by $b_x * N_{b_y} + b_y$.

EXTENSION OF SYSTEM

It should be noted that the FIRE system is an unconventional FIS. The Mamdani FIS is the most common model used for Fuzzy Logic. A wide range of Image Processing operations, not just edge extraction, could be performed using a generalized GPU Mamdani FIS implementation.

The implementation defined in this paper requires only a few modifications to perform generalized Mamdani inference. First, instead of finding the mean value of all antecedent firings for a single rule, only the minimum firing would be used. Second, instead of using only the maximum antecedent firing to perform implication on a single consequent set, all fuzzy rules firings would be used. All that is needed is one global memory segment that is $(N * M) \times R$ in size (the FIRE implementation uses an $(N * M) \times 1$ memory segment, fig 6 c), where R is the number of rules. We showed in [27] that fuzzy consequent set aggregation can be efficiently performed as the first step in the reduction kernel, i.e. no need for a separate pass. This is done by storing the discrete consequents in texture memory, a memory segment of size $S \times R$ (the FIRE implementation uses a memory segment of size $S \times 2$, fig 6 e). In the first step of the reduction kernel for implication, aggregation and defuzzification, each thread samples its corresponding discrete consequent values for the R rules. Next, the minimum of each value fetched and their respective rule firings are computed. The final step is to compute the maximum over these minimums. This is a practical assumption, given that the number of rules in a typical rule base is low, e.g., 32 or less, and performing reduction on a set of elements this small is not ideal on the GPU.

CONCLUSION

This paper described a parallelization of fuzzy-logic based image processing on the GPU. The FIRE edge extractor was demonstrated specifically. Using the NVidia 8800 Ultra GPU,

ISBN: 978-989-95079-6-8

a 126 time speed improvement can be made to the original algorithm making its processing real-time. The GPU can process approximately 42 frames per second at 640x480 resolution using this algorithm.

With such large processing power, more time can be spent on higher level processes. The output of this system could be fed into a larger real-time system that performs more complex operations such as object recognition or tracking. The low price of GPUs and the ease of learning and using the CUDA API make this type of parallel programming a legitimate possibility for a large portion of the programming community.

REFERENCES

[1] K. Pringle. "Visual perception by a computer," *In Automatic Interpretation and Classification of Images*, pages 277-284. Academic Press, New York, 1969.

[2] J. Prewitt, "Object enhancement and extraction", in *Picture Processing and Psychopictorics*, Academic Press, pp. 75-149, 1970.

[3] J. Canny, "A computational approach to edge detection," *in IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 8, pp. 679–698, 1986.

[4] Y. S. Choi and R. Krishnapuram, "A robust approach to image enhancement based on fuzzy logic," *in IEEE Trans. Image Process*, 6 (6), pp. 808-825, 1997.

[5] H. R. Tizhoosh, "Fast fuzzy edge detection," in Proceedings NAFIPS 2002 Annual Meeting of the North American Fuzzy Information Processing Society, pp. 239-242, 27-29 June 2002.

[6] L. O. Hall and A. Namasivayam, "Using adaptive fuzzy rules for image segmentation," *in IEEE World Congress on Computational Intelligence*, Vol. 2, pp. 1560-1565, 1998.

[7] M. Craig and M. Schneider, "On the use of fuzzy sets in histogram equalization," *in Fuzzy Sets and Systems*, Vol. 45, pp. 271-278, 1992.

[8] F. Russo and G. Ramponi, "Edge extraction by FIRE operators", *in Proceedings of the Third IEEE International Conference on Fuzzy Systems*, 1994, pp. 249-253.

[9] L. Zadeh, 'Fuzzy sets,' *Information Control*, pp. 338-353, 1965.

[10] L. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes", *in IEEE Trans. on SMC*, Vol. 3, pp. 28-44, 1973.

[11] E. Mamdani, "Application of Fuzzy Algorithms for Simple Dynamic Plant", *in Proc. IEEE*, Vol. 121, pp. 1585-1588, 1974.

[12] M. Cirstea, J. Khor and M. McCormick, "FPGA fuzzy logic controller for variable speed generators," *Proceedings of the 2001 IEEE International Conference on Control Applications*, pp. 2001.

[13] C. Lynch, H. Hagras and V. Callaghan, "Parallel Type-2 Fuzzy Logic Co-Processors for Engine Management," *Fuzzy Systems Conference, 2007. FUZZ-IEEE 2007. IEEE International*, 23-26 July 2007 pp.1-6.

[14] Hiroyuki Watanabe, Wayne D. Dettloff, and Kathy E. Yount, "A VLSI Fuzzy Logic Controller with Reconfigurable, Cascadable Architecture," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, 1990, pp. 376-382.

[15] K. Moreland and E. Angel, 'The FFT on a GPU,' in *Proc. SIGGRAPH/EUROGRAPHICS Conf. on Graphics hardware*, 112–119, 2003.

[16] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *Intl. Conf. on Computer Graphics and Interactive Techniques*, 908-916, 2003.

[17] S. Larson, C. Snow, M. Shirts, and V. Pande, 'Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology,' *Comp. Genomics,* 2002.

[18] D. Anderson, R. H. Luke, and J. M. Keller, "Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units", in *IEEE Transactions on Fuzzy Systems*, 2007.

[19] D. Anderson, R. H. Luke, and J. M. Keller, "Incorporation of Non-Euclidean Distance Metrics into Fuzzy Clustering on Graphics Processing Units", in *Proceedings, International Fuzzy Systems Association Conference*, 2007.

[20] N. Harvey, R. H. Luke, J. M. Keller, and D. Anderson, "Speedup of Fuzzy Logic through Stream Processing on Graphics Processing Units", *in IEEE Congress on Evolutionary Computation (CEC), WCCI,* 2008.

[21] Y. Mizukami And K. Tadamura, "Optical Flow Computation on Compute Unified Device Architecture," in *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, 10-14 Sept. 2007 pp. 179-184.

[22] L. Pan, L. Gu nad J. Xu, "Implementation of medical image segmentation in CUDA," *Technology and Applications in Biomedicine, 2008. ITAB 2008. International Conference on*, 30-31 May 2008 pp. 82-85.

[23] Y. Luo andR. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPR Workshops 2008. IEEE Computer Society Conference on*, 23-28 June 2008 pp. 1-8.

[24] Nvidia Corp., 'NVIDIA CUDA Compute Unified Device Architecture,' June 23, 2007, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[25] M. Harris, "Optimizing Parallel Reduction in CUDA", NVIDIA Whitepaper, March 18, 2008, http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[26] Nvidia Corp., 'NVIDIA GeForce 8800 GPU Architecture Overview,' GeForce_8800_GPU_Architecture_Technical_Brief.pdf, November, 2006, http://www.nvidia.com/object/IO_37100.html

[27] D. Anderson and S. Coupland, "Parallelisation of Fuzzy Inference on a Graphics Processor Unit Using the Compute Unified Device Architecture", *UKCI 2008, the 8th Annual Workshop on Computational Intelligence*

ISBN: 978-989-95079-6-8