Speedup of Fuzzy Logic through Stream Processing on Graphics Processing Units

Nicholas Harvey, Robert Luke, James M. Keller, Derek Anderson

Abstract—**As the size and operator complexity of a fuzzy logic system increases, computational tractability becomes a problem. There is a significant amount of parallelism in both the creation of the fuzzy rule base and in fuzzy inference. Traditional processors (CPUs) cannot take full advantage of this natural parallelism. Graphics Processing Units (GPUs) speed up rule construction and inference by utilizing up to 128 processing units operating in parallel. Normally, these processors are used to perform high speed graphics calculations for video games, movies, and other areas of intense graphical work. In this paper, a method is discussed for speeding up fuzzy logic by structuring it into a format such that it resembles the standard rendering procedure for a graphics pipeline based on rasterization.**

I. Introduction

The doubling in processing power of CPUs has approximately followed Moore's observation, every 18 months, while GPUs are doubling around every 6 months [11]. Much of this progress is due to the highly parallel nature of graphics applications. GPUs are stream processing devices. Objects on a GPU (referred to as fragments) are processed in parallel by the same program, known as a shader. The number of FLOPS (floating point operations per second) on a stream processing device can be increased, to some extent, by simply adding more processors. GPUs are also optimized for floating point calculations. All of this is a recipe for improved computational performance on problems that can be translated into the stream processing model, such as the translation process and inference procedures of fuzzy logic.

Algorithms based on the independent processing of the elements of vectors and matrices tend to benefit significantly from GPU implementations, as do many other general problems that depend heavily on floating point operations [4]. There is a rapidly growing body of research that focuses on the utilization of GPUs to perform general purpose computation. Some examples include Fuzzy C-Means clustering [1, 2], ray tracing calculations [5], linear algebra [6], and fast Fourier transforms [7]. Because fuzzy logic can be implemented using vector and matrix calculations, it is a prime candidate for GPU optimization.

Manuscript received December 21, 2007. This work was supported in part by the University of Missouri-Columbia College of Engineering.

N. Harvey is with the University of Missouri-Columbia, Columbia, MO 65211 USA (phone: 314-307-4157; e-mail: rhl3db@mizzou.edu, kellerj@missouri.edu, dtaxtd@mizzou.edu)

R. H. Luke, J. M. Keller, D. Anderson are with the University of Missouri-Columbia, Columbia, MO 65211 USA (e-mails: rhl3db@mizzou.edu, kellerj@missouri.edu, dtaxtd@mizzou.edu)

Several fast (real-time) fuzzy logic solutions exist [12, 13]. However, they are typically extremely specialized and designed for control problems. A GPU offers flexibility, as it is more easily integrated with a PC, and speed.

Most applications of real-time fuzzy rule-based systems utilize singleton fuzzification with correlation-min rule encoding [16]. These choices facilitate a simple and efficient implementation. Alternate rule encoding and full inference using Generalized Modus Ponens considerably more costly, but increases the utility of fuzzy systems for decision making purposes. It is this scenario that we address here.

In section 2 we briefly review fuzzy logic. Section 3 contains an overview of GPU programming. Section 4 covers the basics of performing fuzzy logic calculations on the GPU. Sections 5 through 8 describe the packing scheme used for the fuzzy logic system. Section 9 contains technical details of the implementation, and section 10 describes our experiments and contains the results. In section 11, potential future work is discussed.

II. Fuzzy Logic

Fuzzy Logic, put forth by Lotfi Zadeh, is an extension of traditional Boolean logic [9]. This theory is based on fuzzy sets [8], which allow for intermediate truth values, between true and false. Fuzzy Logic is often used in control and decision systems. Truth values are usually determined by using a fuzzy membership function, denoted as $\mu_A(x) \in [0, 1]$, where A is some fuzzy set. Any function that maps onto [0, 1] may be selected as the membership function. Often a discrete membership function is designed, when the inputs are known to be discrete. This type of membership function is simply represented as a vector. Rules in a fuzzy logic system are of the form IF-THEN, where the IF part specifies the antecedents and the THEN part specifies the consequents. Rules are constructed from linguistic variables. These variables take on the fuzzy values or fuzzy terms that are represented as words and modeled by fuzzy subsets of some appropriate domain. An example linguistic variable is temperature, and it can be defined using the terms such as hot, warm, and cool.

First, rule matrices are constructed using implication operators. The implication operator is applied to the antecedent and consequent membership function vectors, resulting in the rule matrix: $\mu_A(x) \rightarrow \mu_B(y) = \mu_R(x, y)$. The compositional rule of inference can then be applied [16], using the rule matrix and an input membership function vector. Inputs to the inference process which are similar to

978-1-4244-1823-7/08/$25.00©2008 IEEE

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

the antecedent will produce corresponding outputs similar to the consequent.

III. Graphics Processor Units

GPUs originally arose from the desire for more powerful computer graphics processing abilities. Prior to the advent of GPUs, graphics processing took place on the CPU; CPUs were primarily designed to execute instructions and perform calculations using integers, not to perform the amount of floating point and vector-matrix operations necessary for most graphical applications. As a result, real-time graphics rendering engines, such as those used by most video games, had severe limitations on the quality of the scenes, and it took a very long time to produce the special effects used in many movies.

In order to improve the speed and quality of graphics, specialized peripheral hardware was created. It was designed solely to perform the mathematical operations used in the rendering of computer graphics based on the theory of rasterization. Rasterization is simply the projection of a scene in a continuous, three dimensional space, onto a discrete, two dimensional display. The vertices in 3-space are grouped into polygons (normally triangles), which are then projected onto the 2-space of the display, and filled as appropriate. Because the hardware could be optimized for this relatively narrow set of tasks, it could perform the tasks much faster than the CPU. As GPUs became more common, the need became apparent for a standardized, platform independent, high level language. GLSL is the OpenGL shading language, designed to integrate with OpenGL. HLSL is Microsoft's high level shading language, and works with DirectX graphics libraries. Cg is a C like language for GPU shader programs, designed by NVIDIA, which works with both DirectX and OpenGL.

As mentioned above, modern GPUs are stream processors. When presented with a collection of input data, such as pixels in an image, stream processor computes a function on the entire collection, much like a kernel operation, where each stream element is assumed to be independent of all others. GPUs typically process data in two stages. The first step is vertex processing. The vertex processors are responsible for transforming the vertices that make up the geometric objects, or simply passing the vertex positions on to the next step: primitive assembly. In this stage the polygonal primitives are constructed from the respective vertex positions. This shape is then rasterized into fragments. Finally, fragment processing is performed, in which the color information of each fragment can be modified by a fragment program, before the scene is displayed.

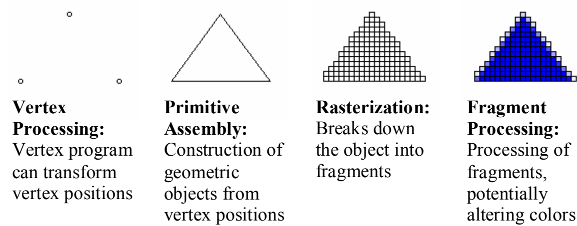

Fig. 1. Rending stages in the graphics pipeline

Only fragment and vertex processors are currently programmable on the majority of commercially available GPUs. The Shader model 4 was recently released, and a new processing stage, geometry processing, has been added. (This stage allows for operations on a primitive after vertex processing and before the object is rasterized.) Because there are almost invariably more fragments in a scene than vertices, virtually all GPUs contain more fragment processors than vertex processors. Since fragment processors have been designed and optimized for per-pixel operations on textures, they are ideal for matrix calculations. Therefore, most GPGPU (General Purpose GPU) programs rely primarily on the fragment processesors and programs in order to perform their calculations.

The rendering process, which takes place on the GPU, is controlled by commands issued from a CPU program. This controlling program must manage the settings of the GPU, load all data, pass the data to and from the GPU, and control the execution of the GPU programs.

The CPU control program uses several libraries to control the GPU. OpenGL is used to control higher-level aspects of the rendering environment. NVIDIA's Cg and CgGL libraries are used to gain access to the GPU and develop GPU programs. Cg controls the profile used during GPU program execution, which determines which features can be used by the GPU program. Different profiles can limit the total number of instructions and arithmetic operations per GPU program, and affect the functionality of conditionals, looping, various data types, array indexing, and Cg standard library functions. These libraries also control the creation, loading, and binding of the GPU programs. These libraries, and their relationships, are expressed in Figure 2.

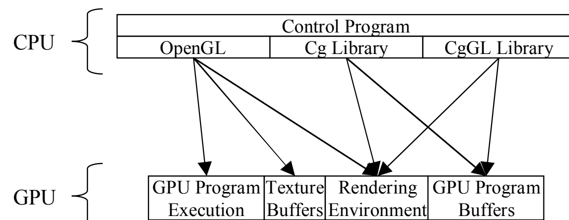

Fig. 2. Interaction between CPU and GPU libraries.

IV. Fuzzy Logic on the GPU

GPGPU programming is a very powerful method for quickly performing complex calculations on streams of relatively independent elements. This processing philosophy takes advantage of the fact that textures stored in GPU

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

memory are simply matrices, typically populated with color information. This means that the GPU can be used to perform calculations which are not graphical in nature, as long as the format of the information is similar to that of a standard graphics application. When the elements of the matrices can be processed independent of the results of their neighbors, speedup can usually be gained from a GPGPU implementation.

Fuzzy membership functions and fuzzy rules are stored as matrices in texture memory, and GPU fragment programs are used to perform operations on these matrices. A continuous membership function must be discretized prior to storage. The size of the consequent membership functions and fuzzy outputs is essentially arbitrary, though they must be within the bounds set by the specific GPU hardware. The NVIDIA 8800 can operate on textures of size 8192x8192, while most other adapters, NVIDIA and ATI, are limited to textures of size 4096x4096. The size of the antecedent and input membership functions must be a power of two, in order to achieve the best performance. When using antecedent and input membership functions with sizes other than powers of two, a significant amount of processing time is wasted. This constraint exists because of a final reduction step. Section 5 outlines rule construction for single rules, with a single antecedent and a single consequent, section 6 shows how to perform inference from Generalized Modus Ponens, section 7 describes the extension to multiple antecedents and multiple inputs, and section 8 shows how to use multiple rules, as in a rule database. All of these configurations were tested on sample data, to ensure that all calculations were being correctly performed by the GPU programs.

V. Rule Construction

Data is transferred between the CPU and GPU using the PCI data bus. In many situations, this can be the most significant performance bottleneck. As a result, it is best to minimize the amount of data transferred between the CPU and GPU, hence only communicating when necessary, such as when the application starts and finishes. Because of this, it is more efficient to store the fuzzy rule set as groups of antecedent and consequent membership functions, instead of the complete rules. Limited memory on the GPU makes the storage of large numbers of pre-calculated rule matrices impracticable, and the amount of time necessary to calculate each rule on the fly is negligible compared to the amount of time that would be needed to copy the rules back and forth between the CPU memory and GPU texture memory each time. In order to select a rule for use in inference, one simply selects antecedent and consequent membership functions, and the rule is calculated. The choice of implication operation is relatively arbitrary; in this case the Lukasiewicz implication, $\mu_R(x, y) = 1 \wedge (1 - \mu_A(x) + \mu_B(y))$, is used.

For a rule with a single antecedent and single consequent, the packing scheme is shown in Figure 3.

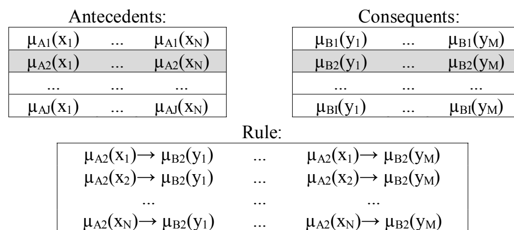

Fig. 3. The construction of a single antecedent rule from the antecedent and consequent fuzzy membership functions

Antecedents are packed into a single texture. This texture is J×N, where J is the number of antecedents, and N is the dimensionality of the fuzzy sets. The consequent texture has the same general format as the antecedent matrix, but it is I×M, where I is the number of consequents and M is the fuzzy set size. The rule matrix is built on the fly using the GPU for a selected antecedent-consequent pair, and it is N×M in size. We discuss later how to pack multiple antecedents into a matrix.

VI. Inference

Once a rule is constructed, inference is performed, and then another rule is created and the process can be repeated. (In section 8 we show how to store and use multiple rules simultaneously.) Inputs are packed into a single texture that is C×N in size (where C is the number of inputs), and has the same format as the antecedent texture. The compositional rule of inference is performed using the new inputs and the currently loaded rule. The result is a set of new outputs, which has the same texture format as the consequent texture, and is of size C×M. Figure 4 shows this specific format.

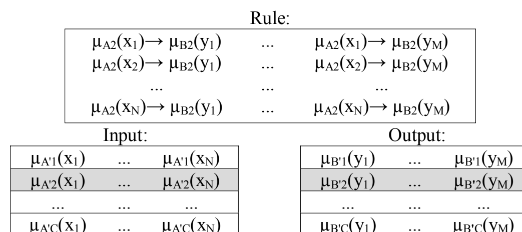

Fig. 4. Performing Fuzzy inference with a single antecedent rule

The first step in GPU inference is to copy the transpose of the input vector into each column of a matrix the same size as the rule matrix. The element-wise minimum of these two matrices is calculated. This is shown in Figure 5.

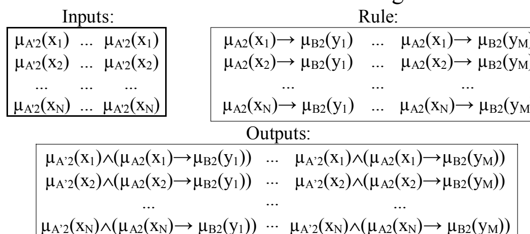

Fig. 5. Element-wise minimum of input and rule matrices

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

Reduction is then applied to shrink the output texture down to a single row. Reduction is a technique commonly used for GPGPU programs; since all information is kept on the GPU hardware during the reduction process, the transfer of data between CPU and GPU is minimized. Reduction simply applies some function, a maximum operation in the case of fuzzy logic, to a pair of values. The resultant value is copied into another matrix, with half the height of the original. The process is repeated until the height is reduced to a single row in the matrix. This vector is the final output from the inference process.

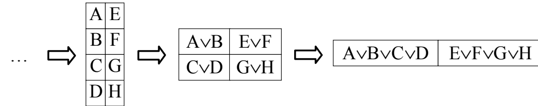

Fig. 6. Vertical reduction of a texture.

VII. Multiple Antecedents and Inputs

Rules with multiple antecedents can be represented by adding dimensions to the rule matrix; a rule matrix with one antecedent is a rectangle (2-D), a rule matrix with two antecedents is a cube (3-D), and a rule matrix with more than two antecedents is a hypercube (>3-D). Alternatively, pages can be set up within the rectangular (2-D) rule matrices, where each page represents a layer in the additional dimensions. With this technique, the same information is stored, only in a slightly different format. Because of difficulties with the literal representation of matrices with greater than two dimensions on the GPU, the latter approach is taken here. The antecedent and input vectors are then combined using a function, usually a minimum. The result is used for rule construction and inference. This procedure is depicted in Figure 7.

We use a slightly different method for combining antecedents and inputs. Rather than building an N×M matrix (Figure 7b), the two vectors are combined into a vector with dimensionality N×M (Figure 7c). This result is used in the same way that the antecedent and input is used for a single antecedent rule construction and inference.

The two input vectors are sampled on the fly, during rule construction and inference to create this N×M vector; that is, $\mu_{A1\times A2}(x_i)=(\mu_{A1}(x_{\lceil i/N\rceil}))\wedge(\mu_{A2}(x_{(i\%N)}))$. Each component in the combined antecedent and input matrices are calculated by taking their position in the single column matrix and mapping that into positions from the original vectors, and the appropriate value, such as the minimum, is calculated.

VIII. Multiple Rule Inference

Multiple rules can be packed into a single matrix in order to have all rules executed in parallel. We first show how this works for a single antecedent. The consequent membership functions for each rule are concatenated into a single vector. Any rule matrices created using this single consequent vector formatting scheme will essentially contain a separate rule matrix for each consequent membership function, in a single matrix format. When the resulting rule matrix is used for

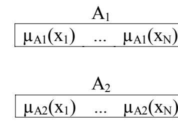

Fig. 7a.

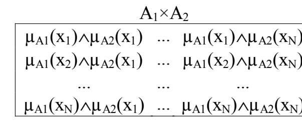

Fig. 7b.

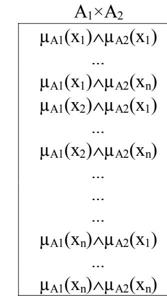

Fig. 7c

Fig. 7. Two methods for combining multiple antecedents for rule construction and inference. We use the second format.

inference, the output vector will contain the inference of each of the stored rules. The outputs can then easily be separated back out of the resulting single row vector. For consequent membership function vectors of size S, the first consequent is located in indices 1 through S of the resulting row vector, the second is in indices S+1 through 2S, the third is in indices 2S+1 through 3S, and so on. The inferences are packed in the same way. This formatting scheme is shown in Figure 8.

The same general procedure can be used to concatenate multiple antecedent clauses together. When combined, these two techniques allow for the construction of a large rule base with multiple antecedents executing in parallel on the GPU.

Unfortunately, a rule is constructed for every antecedent-consequent pair, and many of these rules may not be useful. In order to prevent these rules from influencing the

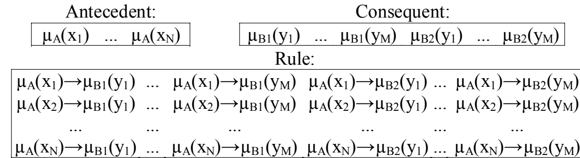

Fig. 8. Storing multiple rules in a single matrix

inference results when fused, their outputs are masked with a single channel matrix of 0 and 1s, allowing the remaining outputs from inference to be combined, while ignoring superfluous data.

IX. Implementation

Because GPU technology varies so much between vendors and series, and substantial advances are being made in very short time periods, it is crucial for the programmer to know the details of the GPU being used, to fully optimize the GPGPU program. The choice of shader language, texture format, rendering profile, data types, and GPU program format, along with other factors, can significantly

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

affect the final optimality of the GPU implementation of the algorithm. Obviously, the model of GPU (and thus the hardware on the GPU) used in the system will have a very significant impact on the speed of the calculations, as will the rest of the system hardware. It is especially important to have the highest possible speed communication bus between the CPU and GPU, as this tends to be one of the primary bottlenecks for most GPU programs.

Because the GPU platform used in this system is NVIDIA hardware, we are using NVIDIA's Cg shader language. The OpenGL texture format used is GL_TEXTURE_RECTANGLE_NV, which allows textures to be non-power-of-two (NPOT) in each dimension. Although the antecedent and input membership functions are required to be a power of two in order to achieve optimal performance, NPOTs permit the consquent and output membership functions to be of arbitrary size. The cgGLGetLatestProfile function is used to have the system automatically select the most optimal fragment profile. All Cg variables are 16-bit floating point data-types (referred to as the half type in Cg). Most GPUs operate faster when using half versus float (32 bit precision) data-types, which makes sense given the precision difference. Additionally, the internal format (for OpenGL) of the textures is GL_RGBA16, because we use a half precision data type. Conveniently, this format automatically clamps to the range [0.0, 1.0], which simplifies many of the fuzzy logic calculations. The GPU programs are designed without any conditional or looping structures. Since CPUs and CPU languages have been heavily optimized for these types of program constructs, and GPUs have not, it is best to avoid their use in GPU programs if at all possible.

The GLUT (OpenGL Utility Toolkit) API is used to provide access to the windowing system, while the GLEW (OpenGL Extension Wrangler) API is used for access to OpenGL extensions. Frame buffer objects (FBOs) are used to render to off-screen destinations, specifically textures, on the GPU. The FBO is a GPU feature that allows for higher speed processing, by keeping data on the GPU as long as possible, and avoiding transfers back and forth to the CPU. The general concept of GPGPU programming is packing data into textures, rendering screen-aligned quadrilaterals that result in a one-to-one mapping of pixels to fragments, and executing fragment programs that perform calculations with the data (fuzzy logic, in this case). One can find a full length, detailed introduction to GPGPU programming at [4]. Figure 9 illustrates our GPU Fuzzy Logic program flow.

X. Results

In order to evaluate the performance gain of GPU Fuzzy Logic, comparison to a traditional CPU program is necessary. This CPU program mimics the actions of the GPU program as closely as possible, while still attempting to take advantage of those features of a traditional CPU which can provide some performance benefit. Each program was timed while it performed identical fuzzy logic calculations. The amount of time used by the GPU program was then compared to the amount of time used by the CPU program.

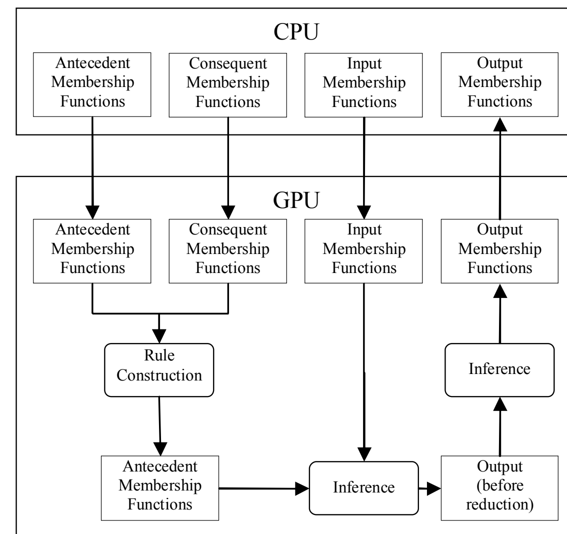

Fig. 9. Program Data Flow

These experiments were performed on a PC, with 4 GB of RAM, a 2.66 GHz, 64-bit, Intel Core 2 Quad CPU, Visual Studio C++ 2005 Express Edition, running Windows Vista as the operating system. The GPU is an NVIDIA GeForce 8800 GTX, which has 128 stream processors, a core clock of 575 MHz, shader clock of 1350 MHz, and is capable of approximately 350 GFLOPS [10]. The times reported below are for the rule construction and inference steps only; they do not include the time spent on initialization for either the CPU or GPU versions of the program. The reason for this is that initialization is done only once, at startup, and can therefore be considered an off-line process. This is a reasonable assumption for any control or general soft-computing system continuously processing data using fuzzy logic.

The size of the rule base is constrained to numbers that we felt represented fairly common configurations, but the number of rules is not limited to these sizes. The number of inputs was varied in the experiments, and large sizes were intentionally chosen. One of our current research topics is continuous video monitoring of elders for well-being assessment and abnormal event detection. This project uses fuzzy inference for reasoning about activities, based on features which are extracted from a three dimensional representation of the human. Image processing and object reconstruction are used to build this representation [14, 15]. We use at least two cameras in each room, in each apartment, and images are captured at 3 frames per second. This generates 10,800 inference inputs for a Fuzzy Logic system to process, for each resident in each room in each apartment for a single hour. There are a large number of inference inputs in this system, relative to the number of rules in the rule base (we currently have 53 rules). Tests of the GPU and CPU Fuzzy Logic implementations were configured to simulate systems of this general structure.

In order to put a large number of inputs through the inference process, multiple passes are used. After each set

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

of inputs is processed, it is read back to main memory, and a new set of inputs is written to the texture memory on the GPU. A similar process is used for the CPU version of the program, except that it is obviously not necessary to read and write to texture memory. The profiles shown below are based on the processing of random data, which was required to achieve the various profiles necessary to show GPU Fuzzy Logic under different workloads. Table 1 reports the performance gain of GPU Fuzzy Logic. Note that all operations are performed simultaneously in all four color channels, thus each input can actually be thought of as four inputs, each rule as four rules, and so on.

The degree of speed obtained by using the GPU is dependent on the exact configuration of the system. In the first experiment, the GPU performed the calculations approximately 154 times as fast as the CPU. In the second experiment, a speedup of approximately a factor of 178 was obtained. And in the third test, an improvement of approximately 190 times was seen. These performance gains are rationalized by the fact that the NVIDIA 8800 has 128 shader processors (potential for around two orders of magnitude gain), the operations performed in GPU fuzzy rule construction and inference are native to the GPU and can be issued in parallel to components packed into an <X,Y,Z,W> vector format (hence operator calculation improvements), and multiple rules can be fired simultaneously through our discussed packing scheme. These performance gains are even greater than those we previously reported for fuzzy c-means (FCM) clustering on a GPU [1, 2], which were slightly over two orders of magnitude, for particular high-throughput clustering profiles.

TABLE 1
RESULTS

Platform:	CPU / GPU
One Antecedent Rules Antecedent Size: 32 Consequent Size: 32 Total Size of Rule Base: 1024 Number of Unmasked Rules: 32 Each of 16384 Inputs Passed Through 32 Rules	164.95 s / 1.07 s = 154.16 s
One Antecedent Rules Antecedent Size: 64 Consequent Size: 64 Total Size of Rule Base: 4096 Number of Unmasked Rules: 64 Each of 131072 Inputs Passed Though 64 Rules	18463.3 s / 26.4 s = 177.53 s
Two Antecedent Rules Antecedent Size: 64 Consequent Size: 64 Total Size of Rule Base: 64 Number of Unmasked Rules: 64 Each of 16384 Inputs Passed Though 64 Rules	162530.0 s / 855.8 s = 189.92 s

XI. CONCLUSIONS AND FUTURE WORK

As demonstrated in this paper, the GPU Fuzzy Logic implementation outperforms the CPU implementation by over two orders of magnitude, under certain configurations. The trade-off for this speed increase is that the GPU implementation is significantly more complex, and there is more time spent on initialization. As with most GPGPU applications, these disadvantages become outweighed by the speedup, with a sufficiently large data set.

The number of rules that are disabled due to masking does not affect performance. In our tests, only a relatively small proportion of the rules were enabled at any one time. With a more efficient rule base organization scheme, however, a larger number of rules could be unmasked and considered active; this would require no additional processing time, but would allow for greater throughput in the inference process.

The next, direct extension to this work is the implementation of a de-fuzzification step on the GPU. Also, as stated above, the tradeoff for the speed gained in using the GPU for Fuzzy Logic is an increase in the difficulty of the The next, direct extension to this work is the implementation of a de-fuzzification step on the GPU. Also, as stated above, the tradeoff for the speed gained in using the GPU for Fuzzy Logic is an increase in the difficulty of the implementation, relative to a program written in a language such as Matlab, or even C. In order to maximize the number of researchers and end users that can use a GPU accelerated Fuzzy Logic system, we plan to use NVIDIA's new CUDA language to write C language programs that are translated to the GPU, bypassing the need to use a shader language such as Cg. The performance is not expected to be as great, but the familiarity of programmers with C is greater.

XII. ACKNOWLEDGEMENTS

This research was partially supported by the National Science Foundation (ITR award llS-0428420) and the National Institutes of Health (5R21AG026412-02). D. Anderson and R. Luke are pre-doctoral biomedical informatics research fellows funded by the National Library of Medicine (T15 LM07089).

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Derek Anderson, Robert H. Luke, and James M. Keller, "Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units," *IEEE Transactions on Fuzzy Systems*, 2007.
[2] Derek Anderson, Robert H. Luke, and James M. Keller, "Incorporation of Non-Euclidean Distance Metrics into Fuzzy Clustering on Graphics Processing Units." Proceedings, International Fuzzy Systems Association Conference, 2007.
[3] Chris Harris, Karen Haines, "Iterative Solutions using Programmable Graphics Processing Units," IEEE International Conference on Fuzzy Systems, 2005.
[4] GPGPU, <http://www.gpgpu.org/>.
[5] T. J. Purcell, "Ray Tracing on a Stream Processor," Ph.D. Dissertation, Standford University, 2004.
[6] J. Krueger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," ACM Transcriptions on Graphics (TOG), vol. 22, no. 3, 2003, pp. 908-916.
[7] K. Moreland and E. Angel, "The FFT on a GPU," SIGGRAPH Eurographics Workshop on Graphics Hardware 2003 Proceedings, 2003, pp. 112-119.
[8] L. Zadeh, "Fuzzy sets," *Information Control*, 1965, pp. 338-353.
[9] L. A. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes," in *IEEE Transactions on System, Man, and Cybernetics*, vol. SMC-3, 1973, pp. 28–44.
[10] NVIDIA Corp., "GeForce 8800," 2006, <http://www.nvidia.com/page/geforce_8800.html>.
[11] J. D. Owens, et al., "A Survey of General-Purpose Computation on Graphics Hardware," Eurographics 2005, State of the Art Reports, August, 2005.
[12] M. Cirstea, J. Khor, M. McCormick, "FPGA fuzzy logic controller for variable speed generators," Proceedings of the 2001 IEEE International Conference on Control Applications, pp. 2001.
[13] 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.
[14] D. Anderson, R.H. Luke, J. M. Keller, M. Skubic, "Modeling Human Activity From Voxel Person Using Fuzzy Logic," *Under Review, IEEE Transactions on Fuzzy Systems,* 2007.
[15] D. Anderson, R.H. Luke, J. M. Keller, M. Skubic, "Linguistic Summarization of Activities from Video for Fall Detection Using Voxel Person and Fuzzy Logic," *Under Review, ComputerVision and Image Understanding,* 2007.
[16] G. J. Klir, B. Yuan, *Fuzzy Sets and Fuzzy Logic,* Upper Sadle River, NJ: Prentice Hall PTR, 1995, pp. 231-236, 327-343.

Authorized licensed use limited to: University of Missouri. Downloaded on January 13, 2009 at 16:43 from IEEE Xplore. Restrictions apply.