In Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture (EAMA),
in conjunction with ISCA 2010

Parallelization of Particle Filter Algorithms

Matthew A. Goodrum,* Michael J. Trotter,* Alla Aksel,+ Scott T. Acton,+
Kevin Skadron*
{mag6x, mjt5v*, alla+, acton+, ks7h*}@virginia.edu
*Department of Computer Science
+Department of Electrical and Computer Engineering
University of Virginia, Charlottesville, VA 22904

Abstract

This paper presents the parallelization of the particle filter algorithm in a single target video tracking application. In this document we demonstrate the process by which we parallelized the particle filter algorithm, beginning with a MATLAB implementation. The final CUDA program provided approximately 75x speedup over the initial MATLAB implementation.

1. Introduction

The optimization of algorithms by means of parallelization is at the forefront of the field of computer science. The main reason for this movement is because of the recent popularity of multi-core processors, which are capable of running multiple independent logical threads at the same time. One of the most exciting types of multi-core processor is the Graphics Processing Unit (GPU). GPUs are normally used for the rendering of 3-dimensional graphics to the computer screen; however, recent developments such as the programming library Compute Unified Device Architecture (CUDA) offer the prospect of using GPUs for general programming [10]. In addition to CUDA, another popular approach to parallel programming is the Open Multi-processing (OpenMP) library, which provides programmers with a set of compiler instructions that make full use of multi-core CPU capabilities [11].

The goal of our technical project is to use multiple styles of parallel programming to increase the efficiency of the particle filter (PF) algorithm, which is a probabilistic model for tracking objects in a noisy environment. We are working together on this project to explore the potential speed increases that can be gained using the CUDA and OpenMP programming libraries.

This paper has three major divisions. First we describe the PF algorithm and its applications in image analysis. We then describe the different approaches we took to parallelization. Finally, we share our results, interpretation of those results, and compare them with other attempts at parallelization of the PF.

2. Particle Filter Algorithm

The PF is a statistical estimator of the state of a target given noisy measurements of the state [2]. In this work, state refers to the position of the target. Using a Bayesian framework, the PF estimates the posterior density by a given set of samples. These samples are known as particles. Each particle has an associated weight, based on a chosen image property, which is used to build a likelihood model [1]. This likelihood model is then used in subsequent time steps. Finally, the weights and the particle locations are utilized to estimate the target location [1].

In image analysis, the PF merits research into parallelization due to its plethora of applications. A majority of these applications lie in the field of feature tracking, in particular, different forms of surveillance from facial recognition [9] to the following of vehicles in traffic [1]. Also of interest is the use of the PF in video compression [15]. An additional application, and one particularly useful to our project, is the tracking of leukocytes (white blood cells) [4]. The problem with most PF implementations, however, is that the computational cost is prohibitive for real-time applications. The intent of our project, then, is to provide enough speedup to the PF algorithm to allow for real-time processing of data. With this ability, the algorithm would allow interactive and immediate results, greatly facilitating its adoption.

3. MATLAB Implementation

We began our work on this project with a version of the PF written in MATLAB. The program was divided into two sections, the first generating a synthetic video sequence for use in the second section which contained the implementation of the algorithm. The video sequence simulates the motion of a white blood cell with additive noise by picking a point in each frame, dilating that point, and then adding random Gaussian noise to the frame. The PF section takes the video sequence as input, with a predefined motion model representing the estimated path that the object will follow. For every frame in the provided video sequence, the algorithm makes one hundred estimations about the location of the object in that frame. These estimations are weighted according to the image, the weights are normalized, and the estimations are updated.

4. Conversion from MATLAB to C

Our first step was to translate the given MATLAB code into C. A majority of this conversion was straight forward line-by-line adaptation; however, the built-in MATLAB functions required significant work due to a relative lack of documentation available. For example, MATLAB provides a Gaussian random number generator, whereas C only contains a Uniform distribution random number generator. We transformed the Uniform distribution to a Gaussian distribution using the Box-Muller algorithm, however this has a drawback. The Box-Muller algorithm uses expensive operations including logarithm, cosine, and the square root function [3]. This cuts back on some speed gains in the C implementation.

Other MATLAB functions provided an added difficulty as it was not perfectly clear what purpose they served in the algorithm. Instead of simply looking at documentation we had to observe the behavior of these functions in order to mimic their functionality. An example of this is the imdilate function, which dilates the image. In the PF, this function is used to expand the size of the object within the video sequence to accommodate the error added by the algorithm.

Before moving on to the parallelization of the program, we measured the degree of auto-vectorization carried out by the compiler. Under optimization level 3 (-O3), the Intel C Compiler (icc) and the GNU C Compiler (gcc) were both making use of SSE and SSE2 instructions. icc was also inserting MMX instructions, while gcc had virtually none of these instructions. In addition icc used SSE and SSE2 instructions with a much greater frequency than gcc. Upon re-compiling to eliminate the use of these mini-vector instructions, we found that there was a negligible impact on execution time. Nevertheless, we use the version of the code incorporating SSE instructions for the remainder of our analyses.

5. OpenMP Implementation

The next step after completion of the C implementation was to parallelize the program using OpenMP. First we began profiling the program to find the sections that dominate the execution time. For the PF, we determined that 90% of the execution time was taken by the section where the estimations are updated according to the normalized weights. Although other sections of the program had data-dependencies, this update section did not, making it a good candidate for parallelization. With this section parallelized the OpenMP implementation provided a 2.5x speedup, with a theoretical maximum increase of 3.0x.

The section responsible for a majority of the remaining execution time was the likelihood calculation. While there were no obvious data-dependencies in this section, there was an array that every thread accessed. This led to a race condition preventing us from getting speed gains. In order to solve this problem we provided each thread with its own copy of the array.

The last section taking up a significant portion of the execution time was the generation of random numbers. The built-in C rand function is not thread-safe because it requires previous calls to rand to ensure that the next call provides a different random number. In order to parallelize this section, we had to provide each thread with its own seed value and create a thread-safe random number generator, specifically the Linear Congruential Generator. This thread-safe LCG, programmed specifically as a CUDA device function, is separately packaged and will be available online.

The remaining loops were relatively basic to parallelize, with OpenMP providing all the needed functionality, including a few parallel sum reductions. With a majority of the program parallelized we reached a maximum speedup of 3.75x.

6. Naïve CUDA Implementation

Knowing that the update portion of the code was responsible for the vast majority of the execution time, we decided to only transcribe this portion of the code in CUDA. In addition, we knew that this section would parallelize more easily compared other sections of the execution, including the normalizing weights and the random number generation portions. These sections

had data dependencies that would require substantial reworking of the code, in order to function in CUDA. This translation entailed writing a single kernel which would be executed for the processing of each frame. In addition, this kernel would have the find index function as a device function that each thread would call as part of the updating sequence. After completing this section, we were able to get speedups on par with the OpenMP implementation.

The issue with this implementation was that every frame required the loading of data back from the GPU to the CPU, and then back to the GPU again. It was obvious that the program would be bottlenecked by I/O if we had to move data back and forth every frame. This is because CUDA memory copy functions require significant overhead, in that they require a global synchronization of all threads. We tried to process a minute of video (1800 frames) with 100,000 particles, but the overhead of moving information back and forth from the GPU was approximately half a second per frame, compared to 0.7 seconds per frame total. With our goal being to process a minute of video with a high number of particles, we knew we had to move the whole program to the GPU.

6.1 Naïve versus Thrust

In addition to the initial naïve CUDA implementation, we also experimented briefly with the Thrust library. Thrust is an abstraction of CUDA for C coding that allows parallel programming on the GPU without using CUDA library functions.

Our experience with Thrust was that it required a substantial reworking of our code in order to provide worthwhile performance. This is mainly due to the fact that Thrust organizes data into vector objects that require iterators to modify individual pieces of that data. A great portion of our code involves complex functions and somewhat irregular accesses, two things that Thrust does not support efficiently.

To be more specific, Thrust applies changes to its data vectors using its built in transform function. The transform function applies a functor to all of the specified data from a beginning iterator to an ending iterator. This works well for simple mathematical tasks, such as multiplication of all data elements by another set of elements or a constant. However, the more intricate functions required in the likelihood calculation require manipulation of data discontinuously. In order to achieve the same result, Thrust requires us to use the “transform_if” function, which along with a stencil vector of 0’s and 1’s, applies a transformation to specific values in another array. It takes three inputs: the array of values to be modified, the stencil vector that specifies which indices to modify, and a functor that represents the transformation to apply. We saw this as likely requiring a large additional overhead. For this reason we decided to just carry on with a full CUDA implementation, and discontinued our work with Thrust.

7. CUDA Optimizations

In order to port the rest of the program onto the GPU, we had to first overcome several obstacles. The normalization of weights required a tree reduction for the summation of the weights. In addition, several sections, including calculation of likelihoods and the random number generation, required thread-specific copies of data structures. The reason for these data-dependencies is that the results of a previous frame are used in the calculations in the next frame.

7.1. Tree reductions

CUDA does not have a simple, straight-forward way of calculating sums in parallel like OpenMP. In order to perform a reduction across multiple thread blocks we had to force a global synchronization of threads with an additional kernel call. The partial sums within each thread block are calculated in parallel using a simple tree reduction algorithm. After a global synchronization the partial sums are added serially.

7.2 GPU Linear Congruential Generator

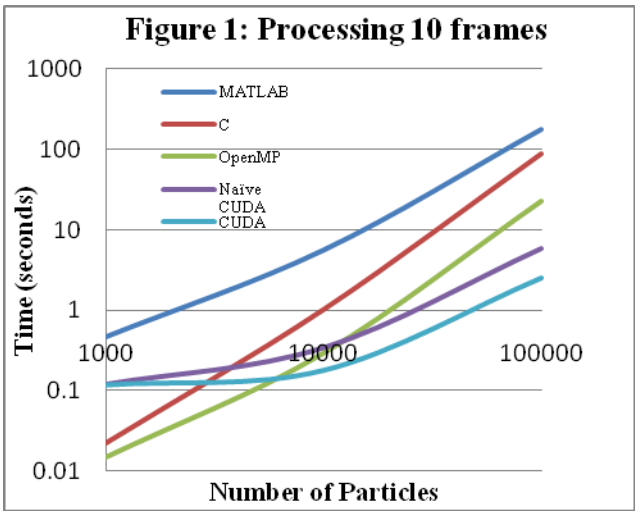
CUDA has no built-in random number generator, but the PF requires both Gaussian and Uniform random numbers every frame. Because our goal was to prevent all CPU-GPU communication within a frame, this meant we had to generate these random numbers either entirely ahead of time or on the GPU. Since we want three random numbers per particle per frame, it quickly becomes unreasonable to move that much data to the GPU ahead of time. Thus, we elected to create a random number generator function for both the Gaussian and Uniform distributions on the GPU.

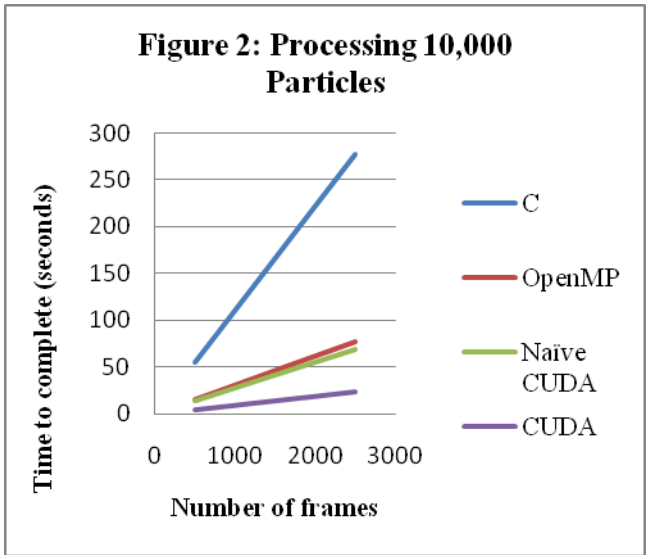
Our implementation uses a Linear Congruential Generator (LCG) which creates uniformly distributed random numbers. We still use the Box-Muller algorithm to convert these random numbers to the Gaussian distribution when needed. In order to make sure that we obtain unique random numbers, we provide a seed value in an array for each thread, using CPU clock values. Each thread updates its own seed value, making accesses to the parallel and thread-safe.

8. Results

We performed several layers of testing in order to find the situations in which parallelization of the algorithm was most beneficial. The machine running the MATLAB, C and OpenMP versions had a Core2 Duo Extreme processor running Ubuntu 8.04.4 LTS. The C and OpenMP version were compiled using GCC 4.2.4 or ICC 11.1. The MATLAB version we used was 7.8.0.347 (R2009a). The CUDA versions of the program ran on a Core i7 and a GeForce GTX 285. The operating system was the same between both computers, and CUDA was compiled using CUDA 2.2.

We chose to parallelize the algorithm by providing individual threads for each particle, because each particle requires information from a previous frame. Increasing the number of particles produces a greater deviation in the execution times of each of our implementations, as can be seen in Figure 1 (we used logarithmic scale to fit the data in the graph visibly).





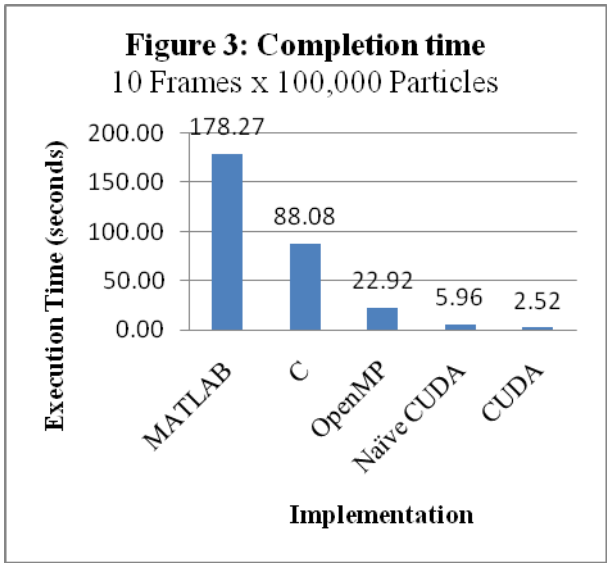
Increasing the number of frames, however, has a smaller effect on this deviation, shown in Figure 2. The MATLAB implementation has issues with run-time as the number of frames exceeds 500, so it was left out of this graph.

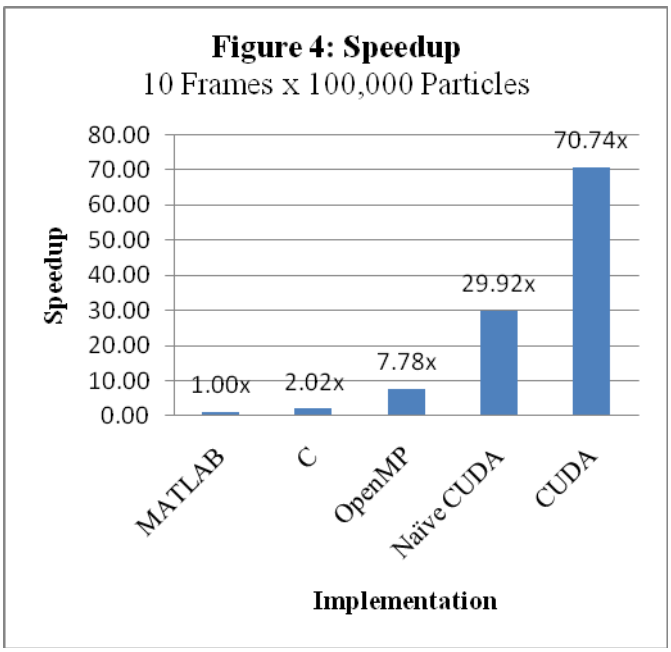
Figure 3 displays the execution times with 10 frames and 100,000 particles, numbers intended to clarify the differences in scaling between implementations of the algorithm. Figure 4 shows the relative speedup values for the same inputs.

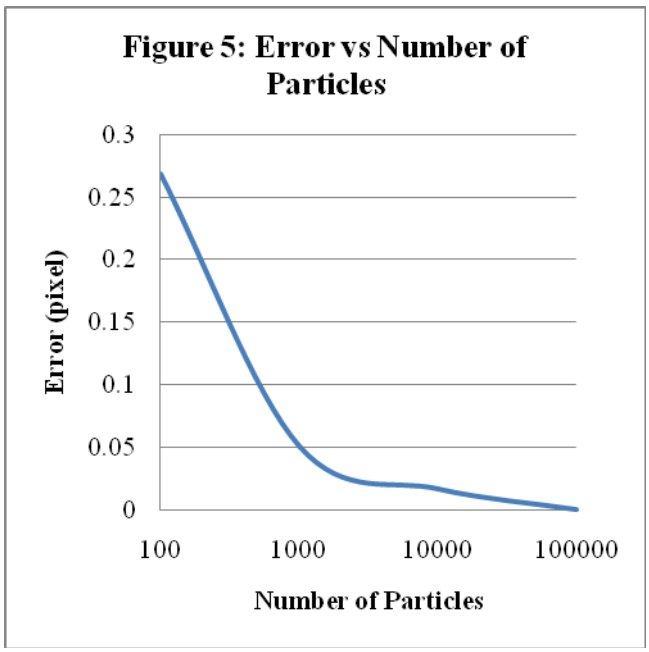
Finally, Figure 5 shows the average error rates, in pixels, which were determined for different numbers of particles. The amount of error and the number of particles are inversely proportional. The upper limit of the reasonable number of particles for a given video sequence is the number of pixels on the screen, because our algorithm does not define the space between pixels. Estimations made between pixels are simply rounded to the nearest pixel. Thus, exceeding the number of pixels on the screen produces no noticeable benefit. For the video sequence we used, the resolution of the screen was 128 by 128; therefore, the maximum number of particles was 16,384.

Since the only way to guess the incorrect position would be to have an error of half of a pixel or more, having an average error rate well below that threshold will almost guarantee successful estimations. For 100 particles, Figure 5 shows an average error rate of over 0.3 pixels, so it is highly likely that with that many particles there will be incorrect estimations. The error rates rapidly drop, however, and 1000 particles produce highly accurate results.

The lesson to take away is that the upper limit of the number of pixels in the image is theoretically useful, but not practically useful. Our full CUDA







implementation is, however, capable of running 10,000 particles at over 55 frames per second, so that theoretical maximum is not an issue on the small images we used..

9. Integration with MATLAB

In order to make the CUDA implementations useful they needed to be callable from MATLAB. This required the creation of a MEX file, which also allowed MATLAB functions to be called from the program. This enables customization of the likelihood function, a feature that is important for optimizing the particle filter accuracy or applying it in other contexts. However, this feature is only available to the naïve CUDA, C, and OpenMP versions of the program, because the optimized CUDA program runs the likelihood function on the GPU. In order to customize the likelihood function in the optimized program, the device function would have to be modified in C, as the GPU cannot call external functions.

10. Related Work

In our survey of the related literature on the parallelization of the PF, we found that the algorithm was being used for a variety of applications, and that this variety was influencing the approach to parallelization. Furthermore, many of these papers describe application-specific optimizations that do not relate well to parallelization of the algorithm in a more general form.

Even though there have been several attempts to parallelize the PF algorithm, many of these attempts focused more on using stream processing [8] or OpenGL [7], rendering them too different for direct comparison. Yet, there are two efforts that are worth discussing, although neither mentions porting the algorithm from MATLAB. Ferreira, Lobo and Dias [5] ported a facial recognition and tracking algorithm to CUDA that implemented the PF for tracking human faces. The focus of their project was real-time robot vision and, consequently, their version of the PF was specialized for that purpose. There is also the work by Ulman [14], in which he optimized a PF algorithm for tracking naval vessels.

Although these projects utilize the main PF algorithm, they differ significantly in their calculation of the likelihoods phase. The reason for customizing this section of the algorithm is that it affects the accuracy of the estimations for the particular application being used. Depending on the nature of the implementation, this can drastically change the approach to parallelization. In our program, the likelihood function is not as complex as the ones described in these papers, lending itself very easily to parallelization.

Other sections of the algorithm were very similar across implementations. In particular, Ulman [14] showed a profile of the execution time of his program that matched ours closely. His program spent nearly 90% of its time updating the weights like ours did with the likelihood calculations taking up the majority of the remaining execution time.

While there are similarities across all implementations of the PF that parallelize easily,

performance can be limited by the properties of the specific application. Therefore, optimizations tailored to the particular application can yield additional speedup, but do not lend themselves to a more general approach to parallelization.

11. Conclusions

As the results in Figure 1 show, the CUDA implementations are slower than both the C and OpenMP implementations until a certain number of particles are used. The naïve CUDA implementation is not faster than the OpenMP implementation until over 10,000 particles are used, while the optimized CUDA implementation is not faster until around 9,000 particles. This is due to the overhead of kernel calls and the copying of data from the CPU to the GPU. C and OpenMP provide very fast execution times under this 10,000 particle count, but the CUDA implementations become significantly faster above this count (the axes of the graph are logarithmically scaled, so at 100,000 particles optimized CUDA is 32x faster than C, naïve CUDA is 13.7x faster than C).

Also important to notice is that increasing the number of frames is a linear increase in execution time. As is apparent in Figure 2, the optimized CUDA implementation has a much lower slope than the others. This is primarily because the data is moved to and from the GPU only once, allowing maximum use of the GPU's acceleration. The more frames that need to be processed the more of a benefit the optimized CUDA implementation gives. The naïve CUDA implementation and the OpenMP implementation are on par with each other for the most part, although the naïve CUDA implementation is slightly faster in general.

Overall, the PF is a highly parallelizable algorithm, and further optimization remains possible through several avenues with the full CUDA implementation. An example of a direct improvement on our work would be to improve the reductions to add block sums in parallel instead of serially. Also, we implemented a binary search for the CDF function, producing generous speedup. Unfortunately, it reduces the accuracy of the algorithm by a marginal amount, so it was not included in our final implementation. Even the current performance increase offers great promise for PF algorithms in real-time video mining.

12. Recommendations for Further Work

As a recommendation for a possible future project, random number generation on the GPU is a major issue for the speed of our algorithm. As was mentioned previously, CUDA has no built-in random number generation, so the expedient creation of random numbers is a primary concern. All Monte Carlo algorithms are dependent on efficient and faithful random number generation, so this would be a great avenue for future work.

For the PF specifically, there are further CUDA optimizations that could be implemented. Improving the tree reductions so that they do not serially add block sums would reduce execution time. Finding a way to remove the need for global synchronization related to the reduction would also provide additional speedup.

Another avenue for further research would be to expand the algorithm so that it can track multiple objects at the same time. This would simply require additional looping and data structures to contain the information for each object. This would open the possibility for speedup from parallelizing frames and objects, instead of particles.

13. Acknowledgements

This work was supported by NSF grant no. IIS-0612049 and a grant from NVIDIA Research.

14. References

[1] Aksel, Alla, and Scott T. Acton. "Target Tracking Using Snake Particle Filter." *2010 Southwest Symposium on Image Analysis and Interpretation.* Austin, TX, IEEE Computer Society, 2010.

[2] Arulampalam, M. Sanjeev, Simon Maskell, Neil Gordon, and Tim Clapp. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing* 50(2):174-188, 2002.

[3] Box, G. E. P., and Mervin E. Muller. "A Note on the Generation of Random Normal Deviates." *The Annals of Mathematical Statistics* 29(2):610-611, 1958.

[4] Boyer, Michael, David Tarjan, Scott Acton, and Kevin Skadron. "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprecessors." *23rd IEEE International Parallel and Distributed Processing Symposium.* Rome, Italy: IEEE, 2009.

[5] Ferreira, João Filipe, Jorge Lobo, and Jorge Dias. "Bayesian real-time perception algorithms on

GPU.” *Journal of Real-Time Image Processing*, 2010.

[6] Gilliam, Andrew D., Frederick H. Epstein, and Scott T. Acton. "Cardiac Motion Recovery via Active Trajectory Field Models." *IEEE Transactions in Biomedicine* 13(2), 2009.

[7] Lenz, Claus, Giorgio Panin, Alois Knoll. “A gpu-accelerated particle filter with pixel-level likelihood.” *International Workshop on Vision Modeling and Virtualization.* Konstanz, Germany, 2008.

[8] Mateo Lozano, Oscar, Kazuhito Otsuka. “Real-time visual tracker by stream processing.” *Journal of Signal Processing Systems*. DOI 10.1007/s11265-008-0250-2, 2009.

[9] Nummiaro, Katja, Esther Koller-Meier, and Luc Van Gool. "An Adaptive Color-based Particle Filter." *Image and Vision Computing* 21:99-110, 2003.

[10] nVidia. "CUDA Reference Manual 2.3." *CUDA Zone*. (July 1, 2009). http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf (accessed October 24, 2009).

[11] Quinn, Michael J. *Parallel Programming in C with MBI and OpenMP*. New York: McGraw-Hill, 2004.

[12] Szafaryn, L. G., K. Skadron, and J. J. Saucerman. "Experiences Accelerating MATLAB Systems Biology Applications." *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits (BiC)*, 2009.

[13] Thrust. *Thrust: C++ Template Library for CUDA.* http://code.google.com/p/thrust/ (accessed April 23, 2010).

[14] Ulman, Geoffrey. “Bayesian Particle Filter Tracking with CUDA.” (April 2010). http://csi702.net/csi702/images/Ulman_report_final.pdf (accessed May 14, 2010)

[15]Wold Eide, Viktor S., Frank Eliassen, Ole-Christoffer Granmo, and Olav Lysne. "Scalable Independent Multi-level Distribution in Multimedia Content Analysis." In *Protocols and Systems for Interactive and Distributed Multimedia*, edited by Fernando Boavida, Edmundo Heitor da Silva Monteiro and Joao Orvalho, 37-48. Heidelberg: Springer Berlin / Heidelberg, 2002.