Image Processing for Multiple-Target Tracking on a Graphics Processing Unit

THESIS

Michael A. Tanner, Second Lieutenant, USAF

AFIT/GCE/ENG/09-11

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCE/ENG/09-11

Image Processing for Multiple-Target Tracking on a Graphics Processing Unit

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Michael A. Tanner, BSCE

Second Lieutenant, USAF

March 2009

AFIT/GCE/ENG/09-11

Image Processing for Multiple-Target Tracking on a Graphics Processing Unit

Michael A. Tanner, BSCE
Second Lieutenant, USAF

Approved:

/signed/	March 2009
Dr. Yong C. Kim, PhD (Chairman)	date
/signed/	March 2009
Lt Col Gregory J. Toussaint, PhD (Member)	date
/signed/	March 2009
Lt Col Michael J. Veth, PhD (Member)	date

Abstract

Multiple-target tracking (MTT) systems have been implemented on many different platforms, however these solutions are often expensive and have long development times. Such MTT implementations require custom hardware, yet offer very little flexibility with ever changing data sets and target tracking requirements. This research explores how to supplement and enhance MTT performance with an existing graphics processing unit (GPU) on a general computing platform. Typical computers are already equipped with powerful GPUs to support various games and multimedia applications. However, such GPUs are not currently being used in desktop MTT applications.

This research explores if and how a GPU can be used to supplement and enhance MTT implementations on a flexible common desktop computer without requiring costly dedicated MTT hardware and software. A MTT system was developed in MATLAB to provide baseline performance metrics for processing 24-bit, 1920×1080 color video footage filmed at 30 frames per second. The baseline MATLAB implementation is further enhanced with various custom C functions to speed up the MTT implementation for fair comparison and analysis. From the MATLAB MTT implementation, this research identifies potential areas of improvement through use of the GPU.

The bottleneck image processing functions (frame differencing) were converted to execute on the GPU. On average, the GPU code executed 287% faster than the MATLAB implementation. Some individual functions actually executed 20 times faster than the baseline. These results indicate that the GPU is a viable source to significantly increase the performance of MTT with a low-cost hardware solution.

To my wife

Acknowledgments

First, I offer my thanks to my wife for her love and support throughout my time at AFIT. She always encouraged me, even when I thought I would not be able to make it through the academic program. And of course, the lunches she packed for me were ever so appreciated. I love you.

I would not have gotten to this point if it were not for my adviser, Dr. Kim. Not only did he teach many of my classes, but he also was always willing to give advice no matter what time I dropped by his office. We spent many hours talking about my research to determine where it would go next. Without his support, I would not have completed this research.

Lt Col Toussaint was the one who helped me learn about multiple-target tracking. He was very patient and flexible with me as we studied original papers and modern textbooks. I greatly appreciated the amount of time and effort he put in to teaching me even though I was the only student in the target-tracking course.

One of the first professors I met at AFIT was Lt Col Veth. He helped me choose my thesis research path, and also suggested a number of courses to provide me with the proper background.

Finally, I would like to thank my fellow research students for making the time at AFIT much more enjoyable than it would be otherwise. Roy, Hiren, and Tom were always willing to lend an ear whenever I was frustrated with my research.

Michael A. Tanner

Table of Contents

Page

List of Figures

List of Tables

List of Algorithms

Algorithm Page

List of Symbols

List of Abbreviations

Abbreviation		Page
MTT	Multiple-Target Tracking	1
CPU	Central Processing Unit	1
GPU	Graphics Processing Unit	1
MEX	MATLAB Executable	2
RGB	Red Green Blue	5
CCL	Connected Component Labeling	6
NN	Nearest Neighbor	12
GNN	Global Nearest Neighbor	12
LR	Likelihood Ratio	14
KF	Kalman Filter	14
FOGM	First-Order Gauss-Markov	16
GPGPU	General Purpose Computing on GPUs	17
CTM	Close To Metal	17
CUDA	Compute Unified Device Architecture	18
OpenCL	Open Computing Language	18
CPI	Clocks Per Instruction	18
ALU	Arithmetic Logic Unit	18
SIMD	Single Instruction Multiple Data	19
SIMT	Single Instruction Multiple Thread	23
GpuCV	GPU Computer Vision	24
OpenCV	Open Computer Vision	24
FRBD	Fast Radial Blob Detector	26
FPS	Frames Per Second	31
JIT	Just-in-Time	33
MHT	Multiple Hypothesis Tracking	45

Image Processing for Multiple-Target Tracking on a Graphics Processing Unit

I. Introduction

Multiple-target tracking (MTT) is a well-defined problem that has been researched since the 1970's. Due to recent availability of MTT related modules in MATLAB or similar high-level programming environments, MTT implementation is not as difficult as it once was.

Since the image processing portion of MTT is such a computationally-intensive process, much research has been done to optimize code and algorithms. Some approaches include using signal-processing chips, custom hardware, or expensive central processing units (CPUs) for marginal performance gains. Money and time can be saved if a relatively inexpensive graphics card can be used to improve image processing performance.

Running a high-level programming language implementation of MTT on a serial processor, like a microcontroller or CPU, has the limitation of handling only serial data. They lack the parallel support needed to fully optimize MTT. On the other hand, a graphics processing unit (GPU) is poor at processing data serially, but is very efficient at solving highly-parallel problems. Using a high-level language that is able to execute code on both a CPU and GPU could, if done correctly, improve the performance of a MTT implementation. Parallel portions of the code (i.e., image processing) can be efficiently executed by the GPU, while the CPU handles the serial algorithms.

1.1 Research Goals

The main goal of this research is to determine if the performance of image processing for MTT can be improved by using a GPU. In order to achieve this goal,

a number of intermediate steps need to be accomplished. The MTT software first must be implemented in a high-level mathematical language. After this, the image processing functions can be converted into C code and then into code that runs on the GPU. Each progressive stage of development should be perform better (faster execution speed) than the previous.

The high-level mathematical language used will be MATLAB, an ideal choice since it is used commonly as an algorithm prototype language. The MTT software must be able to take a series of input images, find the targets within the images, track them, and output the results. All important parameters within the software should be easily accessible to simplify modifications. Also, the code must be well documented in order to aid in future research using the same software. The speed of this software implementation will be the baseline by which all other implementations are compared.

Second, the slowest portions of the MTT image processing code will be converted into C code. The C code can then be compiled to integrate with existing code as a MATLAB executable (MEX) file, thereby removing the MATLAB computing overhead.

Finally, the image processing functions will be converted to execute on the GPU and the performance will be compared with the MATLAB and C implementations.

1.2 Assumptions

A number of assumptions are made to facilitate the development of the MTT software. First, the sensor used by the MTT software is an imaging device (video or still camera). The data processed by the software is 24-bit RGB color images. Assuming the images are in RGB format is necessary to correctly parse and process the inputs (e.g., convert from color to grayscale).

The frame rate from the camera must be constant (i.e., not variable-rate video compression). This simplifies the calculations to predict and update target locations.

The camera is stationary to make the image processing more manageable. On a moving platform, an optical-flow algorithm would have to be implemented to create a common coordinate system for all sequential video frames. Implementing such an algorithm goes beyond the scope of the research goals of this thesis.

Finally, the target motion tracked by this system is assumed to be approximately linear. This makes the filtering and prediction more tractable for the MTT application.

1.3 Contributions

There are two contributions this research provides. First, it delivers a self-contained MTT software system that is well-documented so future researchers do not have to redevelop the code. Also, this research determines if augmenting MTT image processing systems with a GPU improves performance. If so, more research can be done to find other MTT applications for GPUs.

1.4 Thesis Organization

This thesis is organized into five chapters. This first chapter provides a brief overview of the problem, states the research goals, and also explains any major assumptions made in the research. Chapter II details background material needed to understand the research. In particular, it explains image processing, MTT, and GPU programming. A brief summary of related work is also provided. Chapter III explains how the MTT software was developed. It explains how the different stages of the software work and explains the specific algorithms used or developed for each implementation. Chapter IV analyzes the performance of the software. The tests for the software are described, and the performance of each implementation is provided. In-depth reasons are provided for any results that are not intuitive. Finally, Chapter V summarizes the entire research project and suggests future research ideas.

II. Background

A few fundamental concepts must be covered to properly understand this research. There are three general areas of study this thesis builds on: (1) image processing, (2) multiple-target tracking, and (3) graphics processing units. Sections 2.1, 2.2, and 2.3 cover these areas, respectively. This chapter concludes with a brief overview of similar research in the literature.

2.1 Image Processing Background

The goal of image processing in MTT is to take an input from a sensor and produce a list of potential targets and their attributes. For example, the sensor input may be an image from a camera and the list of target attributes may be the centroid, area, and bounding box associated with each target in the image. Figure 2.1 shows the flowchart process for a simple MTT image processing algorithm. This section explains how each colored box in Figure 2.1 works.

2.1.1 Color to Grayscale Transformation. The input is a 24-bit color image, which means it has 8-bits for the red, green, and blue intensity of each individual pixel [1]. Using this encoding method, a total of 16.8 million different colors can be represented in each pixel. Images stored in this format are called true color because they can represent nearly all the visible colors seen in nature.

Many real-time and embedded image processing systems handle only grayscale or binary images since color image algorithms are too computationally expensive. Therefore, the first step is to convert this color image into a grayscale image. Grayscale images only contain overall intensity information about the original image (in shades of gray). Equation 2.1 shows the calculation used to convert a color pixel to grayscale intensity. The scaling constants for each color component are taken from the `rgb2gray` function in the MATLAB Image Processing Toolbox. The output grayscale pixel is only 8-bits with

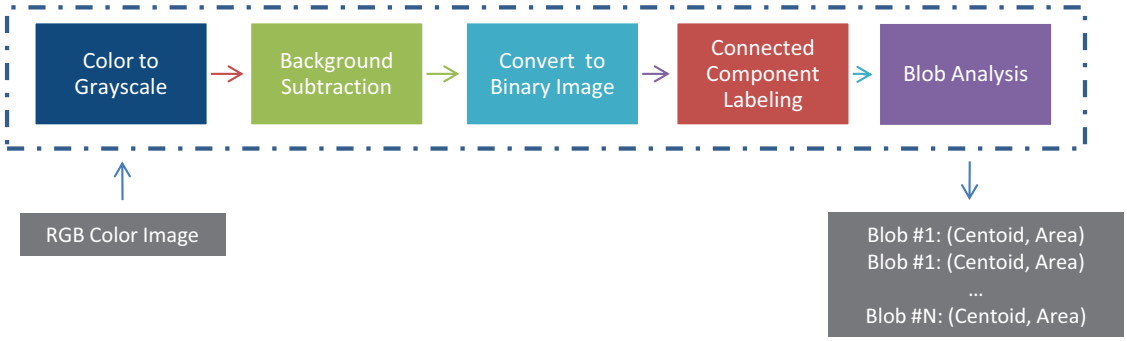

Figure 2.1: The image processing chosen for this MTT application takes five stages. A single color image input is processed, and the final output is a list of the attributes calculated for each potential target found in the image. The attributes calculated are area and centroid.

$$p_g = 0.2989 \cdot p_{c_r} + 0.5870 \cdot p_{c_g} + 0.1140 \cdot p_{c_b} \tag{2.1}$$

where p_g is the grayscale pixel, p_c is the color pixel with red, green, and blue (RGB) color components p_{c_r}, p_{c_g}, and p_{c_b}, respectively.

2.1.2 Background Subtraction. Removing the background from a grayscale image results in an image that only contains light gray pixels for portions of the picture that have changed (i.e., potential targets). Conversely, dark gray pixels in the new image are portions that have not changed. There are a variety of methods used to subtract the background, ranging from simple methods like frame differencing to more complex but robust methods such as background modeling.

The simplest method of background subtraction is frame differencing [2]. Assuming a background image is available (or can be calculated), this method finds the absolute difference between the background image and the current image. Equation 2.2 shows the calculation for each pixel.

$$p_d = |p_{g_1} - p_{g_2}| \tag{2.2}$$

where p_d is the grayscale difference pixel, p_{g_1} is the grayscale pixel calculated in Equation 2.2, and p_{g_2} is the grayscale background pixel.

A slightly more robust method for subtracting the background is N-frame differencing [3]. The average of the previous N video frames is subtracted from the current video frame. This requires more computation, but it is well suited for tracking slow-moving targets relative to the frame rate.

Many other methods have been developed for estimating the background including running Gaussian average, temporal median filter, and kernel density estimation. Detailed descriptions of these methods are available in [4].

2.1.3 Convert to Binary Image. The next step is to convert the current grayscale image (with the background removed) into a binary image. A binary image is purely black and white; it can be thought of as an image where each pixel either is part of a target or is not.

The key to converting the image is to choose an 8-bit threshold value. Each pixel in the grayscale image is compared to the threshold value, if it is less than that value then it is not part of a target, otherwise it is part of a target. Equation 2.3 presents this mathematically.

$$p_b = \begin{cases} 1 & p_d \geq b_{th} \\ 0 & p_d < b_{th} \end{cases} \tag{2.3}$$

where p_b is the binary threshold pixel, and b_{th} is the binary threshold value (between 0 and 255).

2.1.4 Connected Component Labeling. Connected component labeling (CCL) is the process of assigning each potential target, or blob, in the image a unique label, as shown in Figure 2.2. A *blob* is a group of connected identical pixels within a binary image. This labeling is essential in the next step (Section 2.1.5) in image processing, blob analysis. CCL is a well-known graph problem with a number of so-

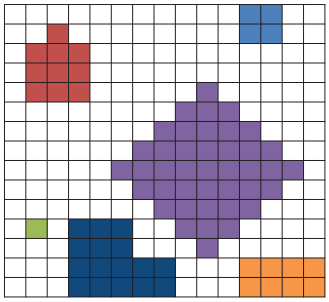

Figure 2.2: The goal of CCL is to assign a unique label to each blob. In this figure, the unique labels are represented as different colors.

lutions [5] [6] [7]. These solutions generally fall into one of two categories, multi-pass and two-pass algorithms [7].

2.1.4.1 Multi-Pass Algorithms. In these algorithms, each pixel in the binary image is processed in sequence. If the pixel is part of a blob and has not been assigned a label, then it is assigned the minimum label of its neighbor pixels. In a 4-connected solution, only the cardinal neighbor pixels (north, south, east, and west) are processed; an 8-connected solution processes all eight neighbors. If a pixel has no labeled neighbors, then it is assigned with a new unique label.

These algorithms continue to loop over the image until no pixels have changed their label. This is the simplest but most inefficient algorithm because it can require a significant number of iterations to propagate all of the labels.

2.1.4.2 Two-Pass Algorithms. Two-pass algorithms use a method called equivalence labeling. On the first pass, each pixel is assigned a new unique label or the minimum of its neighbors as in the multi-pass algorithm. Before moving on to the next pixel though, an equivalence between the neighboring pixels is stored in an equivalence set (typically implemented in a union-find structure). An equivalence means that two labels are synonymous. For example, imagine there are two pixels in a row that are part of the same blob. If they are labeled a and b, respectively, then

Table 2.1: Table to calculate the x-coordinate of the centroid for the blob in Figure 2.3. The final weighted sum should be divided by the area (in pixels).

Column	**Number of Pixels**	**Product**
1	0	0
2	2	4
3	0	0
	Sum:	**4**

Table 2.2: Table to calculate the y-coordinate of the centroid for the blob in Figure 2.3. The final weighted sum should be divided by the area (in pixels).

Row	**Number of Pixels**	**Product**
1	0	0
2	1	2
3	1	3
4	0	0
	Sum:	**5**

the labels a and b are equivalent since they are in the same blob. After the first pass is completed, then each pixel is assigned the minimum label of its equivalence set.

These algorithms tend the be the most efficient [7]. The performance bottleneck of two-pass algorithms is the random memory access caused by set logic. Each time two label sets are marked as equivalent, their lists need to be merged in dynamic memory. Also, on the second pass each label must be found by searching through the equivalence lists. Dynamically allocating, deleting, and traversing heap memory takes significantly more time than using simple stack-based data structures.

2.1.5 Blob Analysis. The final stage of image processing is to perform a blob analysis on the image [6]. The centroid (geometric center) and area of each blob are calculated. Calculating the area is straightforward. After only one pass over the image, the area for each blob can be calculated. While iterating through the image, the blob's area is incremented when the pixel and blob have the same label.

Calculating the centroid can also be completed in one pass, but it requires more calculations [8]. For each blob, a table is generated that stores the number of pixels

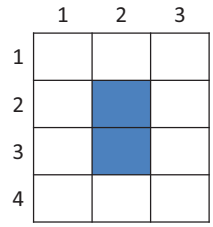

Figure 2.3: Blob analysis is the process of calculating attributes for blobs within an image. In this simple image, the blob has a centroid located at (2, 2.5) with an area of 2 pixels.

the blob has in each row and column. Tables 2.1 and 2.2 provide an example of how these tables are built for the image in Figure 2.3. The number of pixels in each row/column must be weighted by the index of that row/column. Then the sum of all the weighted values is calculated. Finally, this sum is divided by the area to calculate the x- and y-coordinates for the centroid.

For example, the final sum for the rows in Figure 2.3 is 4 with a total area of 2. Therefore the x-coordinate is $4/2 = 2$. Following the same process for the columns will result in an y-coordinate of 2.5. Therefore the centroid is located at (2, 2.5).

2.1.6 Example of Image Processing. A real example is useful to fully understand the steps in image processing. Figure 2.4 provides real images and data from one frame in a MTT video sequence. The figure is structured identically to Figure 2.1 to show the real image that is associated with each image processing block.

The input is a simple color image, and the final output is a list of coordinates and centroids for each blob detected in the image. In order to produce this output, the input image is first converted to grayscale to make the processing more tractable. Then the background is subtracted to determine what has changed in the scene. Next, a threshold is used on the image to mark each pixel as unchanged or changed (i.e., black or white). In CCL, the blobs in the binary image are labeled with unique identifiers. The labels are represented as colors in this example. Finally, the blobs are analyzed to find their centroids and areas.

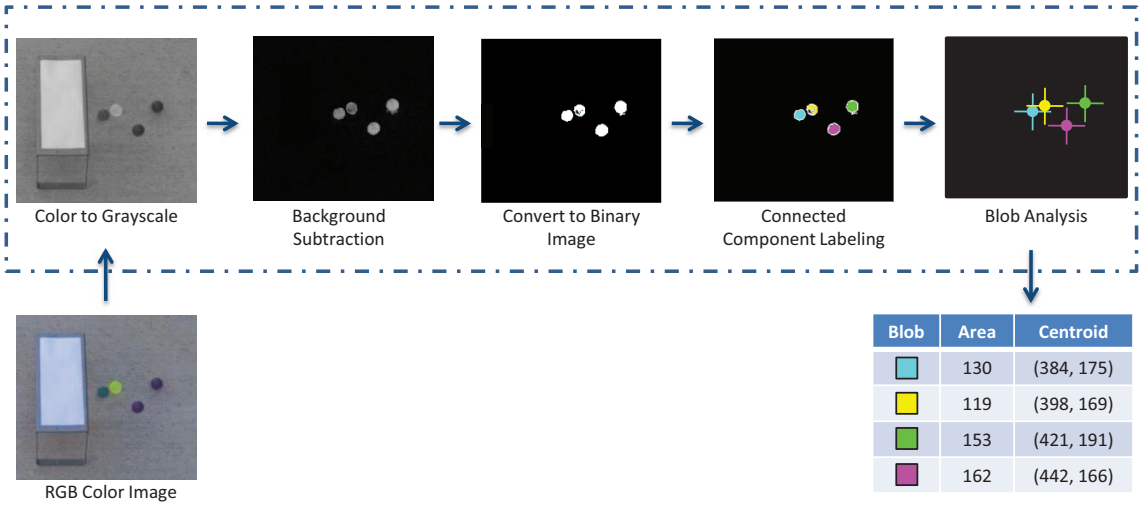

Figure 2.4: This figure is analogous to Figure 2.1, but instead of a generic block representing each stage, the real image/data is shown. As can be seen, the input color image is processed, and the final output is a list of the coordinates and centroids for each blob. The blob labels are represented as colors in this figure for simplicity and clarity, in a real application they would be numbers.

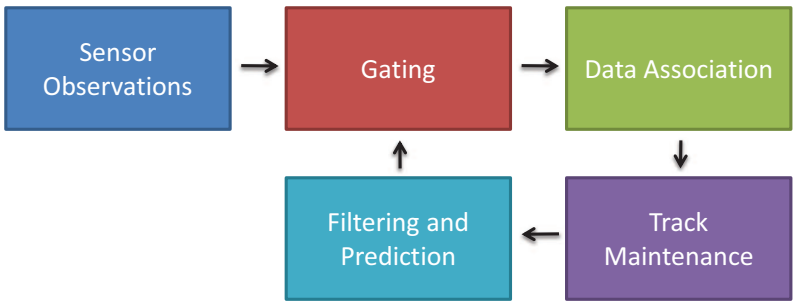

Figure 2.5: A block diagram depicting the recursive flow for a simple modern MTT system. This assumes that tracks have been formed before gating occurs. [9]

2.2 Multiple-Target Tracking (MTT) Background

MTT is a long-standing problem that has been well-researched in the literature. This section will provide a very brief overview of the different stages of a simple modern tracking system, as outlined in Figure 2.5. More in-depth information can be found in [9].

There are two basic terms related to MTT that must be understood before covering the individual blocks in Figure 2.5. An *observation* is one data point that

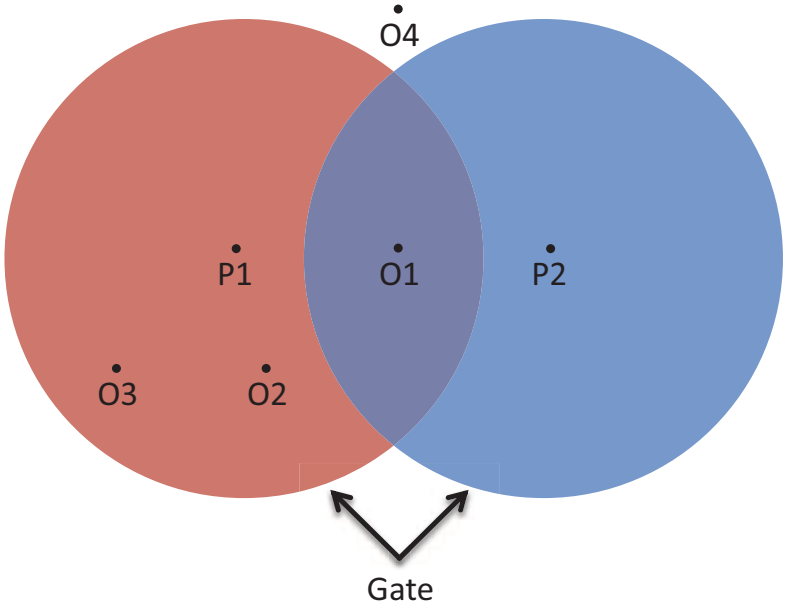

Figure 2.6: A simple gating example with two predicted target locations (P1, P2) and four observation locations (O1, O2, O3, O4). [9]

represents the location of a potential target. In order to track a target over time it is necessary to create a *track*, which is a historical record of observations and predictions associated with a particular target.

Unless otherwise stated, all the information for this section is from [9].

2.2.1 Sensor Observations. The first step of any tracking system is to observe the environment with one or more sensors. If multiple sensors are involved, those data need to be combined (called *data fusion*) into one coherent format. The ultimate goal of this stage is to produce a list of locations where potential targets have been observed.

2.2.2 Gating. Modern tracking systems can cover a large area with many different targets. In order to increase the accuracy of the tracks and decrease the computational complexity of data association, the number of potential observations per target must be limited. Gating is the process of selecting observations that are statistically close to the predicted target location.

A simple way to understand this process is to imagine data from an imaging target-tracking system. In Figure 2.6, two target tracks exist from previous sensor data and MTT loop iterations. There are four observations, but only two of them will ultimately be assigned to the targets. In order to remove unlikely observations, a gate is drawn around each predicted target location. Any observations inside the gate (represented by colored circles in Figure 2.6) can be assigned to that track, while any observations outside the gate cannot.

The size and shape of a gate are important. If a Kalman filter implementation is used for the filtering and prediction, then the covariance matrix can aid in calculating the gate size. The most common gate shapes are rectangular and ellipsoidal. The rectangular gates are simpler to implement, but ellipsoidal gates are more accurate.

2.2.3 Data Association. This is the most important and difficult stage of target tracking. Data association is the process of selecting an observation to be assigned to each track. Sometimes it is appropriate for no observation to be assigned to a track; for example, when a target moves under a bridge it cannot be observed by an airborne camera. One of the difficulties is when observations fall within the gates of multiple tracks (e.g., O1 in Figure 2.6). There must be a way to calculate which track is the best match. A number of different data association algorithms have been developed to solve this problem, including nearest neighbor (NN), greedy algorithm, and global nearest neighbor (GNN).

The simplest solution is using an algorithm called NN. In this method all tracks are assigned to the observation closest to them. Under this scheme, the same observation can be assigned to multiple tracks.

A greedy algorithm performs slightly better. Each track is processed one at a time, assigning it the closest observation that has not already been assigned to a track. This algorithm still is not accurate enough for many modern tracking systems.

One of the most widely used algorithms is GNN. This method seeks to find the most likely association of observations to tracks. It does so by minimizing the

Table 2.3: This table provides an example of an assignment matrix. The columns represent the observations, the rows represent tracks, and each cell represents the statistical distance between that track and observation. [9]

Tracks	O1	O2	O3	O4
T1	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$d_{1,4}$
T2	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$d_{2,4}$
T3	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$	$d_{3,4}$

overall distance between track-to-observation assignments. While this seems to be a straight-forward problem to solve, finding an optimal solution is very difficult. One elegant solution found to date is called the Auction Algorithm which [9] covers in detail.

The Auction Algorithm requires an assignment matrix as the input. Each column in the matrix represents one observation, while each row represents one track. The matrix cell value $d_{M,N}$ represents the distance between track M and observation N. If an observation is outside the track's gate, then it is assigned an infinite value. An example matrix is provide in Table 2.3.

Many additional methods of solving the data association problem exist. Detailed descriptions of such additional methods are provided in [9].

2.2.4 Track Maintenance. Track maintenance refers to the process of creating, confirming, and deleting tracks. A simple and commonly used method is to create a track for every observation that is not assigned to a track after gating and assignment. Once created, there are two different methods used to confirm and delete this track.

A sliding window is commonly used for track maintenance. In this method, the track is confirmed if it has been observed K_c of the last M observations, where K_c and M are parameters set by the tracking engineer. Larger values with a high ratio for $K_c : M$ decrease the number of false tracks, but they also increase the time it takes to confirm a track. Tracks are deleted if they have not been observed in the last K_d frames. While this method is simple, it is effective.

Scoring is a more complicated and usually more accurate approach to track confirmation and deletion. In this method a probabilistic likelihood ratio (LR) is defined to be the ratio between the probability the track is a true track and the probability it is a false track. A track is confirmed once the LR rises above a predefined threshold level T_1, and it is deleted when it falls below a predefined threshold level T_2.

2.2.5 Filtering and Prediction. As can be inferred from the name, filtering and prediction is a two-stage process. Filtering is the process of statistically combining the predicted state of the target with its observed state. Prediction is the process of propagating that filtered state one step forward in time. If no observation is assigned to a target, then the filtering stage is skipped, but the target state estimate is still propagated.

2.2.5.1 Kalman Filter. A number of different methods exist for filtering and predicting, but the most common is the Kalman filter (KF) [9]. The KF is a linear estimator. It is able to optimally combine observations and predictions into an overall optimal estimate of the actual state of a given system (e.g., a target). Along with a state estimate, the KF also has a covariance matrix which represents the uncertainty of its state estimate.

A KF only works as well as its model represents reality. A model is a set of differential equations that describe how a system works. The KF uses current system state information to estimate a future state. Equation 2.4 shows how to propagate a discrete model's state one step forward in time.

$$\mathbf{x}(k+1) = \Phi\mathbf{x}(k) + \mathbf{q}(k) + \mathbf{f}(k+1|k) \tag{2.4}$$

Here, $\mathbf{x}(k)$ is the state vector, Φ is the state transition matrix, $\mathbf{q}(k)$ is the zero-mean, white, Gaussian noise with a covariance matrix Q, and $\mathbf{f}(k)$ is the deterministic input.

In order to update the model with a target's location, a measurement must be taken, as shown in Equation 2.5.

$$\mathbf{y}(k) = H\mathbf{x}(k) + \mathbf{v}(k) \tag{2.5}$$

where $\mathbf{y}(k)$ is the measurement vector, H is the measurement matrix, and $\mathbf{v}(k)$ is the zero-mean, white, Gaussian measurement noise with a covariance matrix R.

Once the model has been created, as in Equation 2.4, and a measurement has been taken, as shown in Equation 2.5, the first KF step is to use Equation 2.6 to update the model with the measurement.

$$\begin{aligned} \hat{\mathbf{x}}(k|k) &= \hat{\mathbf{x}}(k|k-1) + K(k)\left[\mathbf{y}(k) - H\hat{\mathbf{x}}(k|k-1)\right] \\ K(k) &= P(k|k-1)H^T\left[HP(k|k-1)H^T + R\right]^{-1} \\ P(k|k) &= [I - K(k)H]P(k|k-1) \end{aligned} \tag{2.6}$$

where $K(k)$ is the Kalman gain, $P(k)$ is the process covariance matrix, and I is an identity matrix.

Finally, the last KF step is to propagate the estimate forward in time with Equation 2.7 and then repeat the process (update and propagate).

$$\begin{aligned} \hat{\mathbf{x}}(k+1|k) &= \Phi\hat{\mathbf{x}}(k|k) + \mathbf{f}(k+1|k) \\ P(k+1|k) &= \Phi P(k|k)\Phi^T + Q \end{aligned} \tag{2.7}$$

More advanced filters (e.g., extended Kalman filter, unscented filter, particle filter, etc.) can provide better estimates for nonlinear systems, however they are beyond the scope of this research.

2.2.5.2 Singer Model. Any filter and prediction method is only as good as the model it uses. A model is a set of differential equations used to predict the state of a system (with known inputs) over time.

A very popular model used in target tracking systems is the Singer model [10]. It is a linear model that can handle targets with different performance envelopes (e.g., commercial airplane versus military fighter). Since acceleration is modeled as white noise, constant velocity tracking performance is degraded. However, this acceleration model quickly detects target maneuvers.

The Singer model uses the simple one-dimensional state vector shown in Equation 2.8. Tracking a target in two or three dimensions is possible by using two or three decoupled models. Since the models are decoupled, this approach does not take advantage of the relationship between the states of the different dimensions. For example, in a coordinated turn, the x and y states can be better predicted by using knowledge of the radius of the turn.

The second part of Equation 2.8 shows the differential equations that describe the motion of the target over time. The maneuvers are modeled as a First-Order Gauss-Markov (FOGM) process.

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} p & v & a \end{bmatrix}^T \\ \dot{\mathbf{x}} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -\alpha \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} w(t) \end{aligned} \tag{2.8}$$

where p is the position, v is the velocity, a is the acceleration, α is the reciprocal of the maneuver time constant, and $w(t)$ with a covariance matrix Q,

$$Q \approx \begin{bmatrix} T^5/20 & T^4/8 & T^3/6 \\ T^4/8 & T^3/3 & T^2/2 \\ T^3/6 & T^2/2 & T \end{bmatrix} \tag{2.9}$$

where T is the Singer KF sample period.

The state transition matrix in Equation 2.4 simplifies to a Newtonian matrix when αT is sufficiently small.

$$\Phi = \begin{bmatrix} 1 & T & T^2/2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

The KF can be initialized by Equation 2.11 after a target has been observed at least twice.

$$\begin{aligned} \hat{\mathbf{x}}_0 &= \begin{bmatrix} y(1) \\ [y(1) - y(0)]/T \\ 0 \end{bmatrix} \\ \mathbf{P}_0 &= \begin{bmatrix} \sigma_R^2 & \sigma_R^2/T & 0 \\ \sigma_R^2/T & 2\sigma_R^2/T^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{2.11}$$

where y is a vector with the observed positions for the target, and σ_R^2 is the variance of the measurements.

2.3 Graphics Processor Unit (GPU) Background

The parallel nature of GPUs make them more efficient to solve a large set of scientific and engineering problems. Traditionally, GPUs have been very difficult to use for general purpose programming, called general purpose computing on GPUs (GPGPU). In order to be solved on the GPU, algorithms had to be reduced into a sequence of polygon translations and rotations. Random memory access for reading or writing was not allowed.

2.3.1 Programming Languages. Recently two main graphics card manufacturers, ATI and NVIDIA, released software development kits that allow a programmer to access the GPU's power using general purpose code with random memory access rather than formulating problems in the graphics paradigm. ATI called their solution Close To Metal (CTM), which has since been renamed Stream SDK [11]. NVIDIA's

solution is called Compute Unified Device Architecture (CUDA) and currently dominates the GPGPU market [12].

Stream SDK and CUDA are vendor-specific solutions to the GPGPU problem. Apple saw the potential to bridge this gap and pushed for an open standard to be created which would allow GPGPU programming for any vendor. On December 8, 2008 the specification for Open Computing Language (OpenCL) 1.0 was released by the Khronos Group (who also created the OpenGL standard). OpenCL standardizes a GPGPU language, based on the modern dialect of C, which will allow the same code to be executed on any type of GPU. Apple is releasing an implementation of OpenCL in Mac OS 10.6, expected out in mid to late 2009.

2.3.2 GPU versus CPU. CPUs are highly optimized for serial processing of data. In general, the CPU processes one command at a time and operates on only one unit of data. Large amounts of die area are dedicated to control logic and cache to decrease the overall clocks per instruction (CPI). The memory cache is able to prefetch data because of spatial and temporal locality within the code. Elaborate control logic allows instructions to be executed out-of-order and branch logic to be predicted. Very little overall die space is dedicated to arithmetic logic units (ALU) since only one thread can execute at a time. The most modern CPUs (with up to 16 cores on a chip) simply create multiple copies of this basic architecture on one chip.

GPUs take a significantly different approach to computing. Instead of using a large area for cache and control logic, GPUs have very small control logic and cache blocks for a large number of threads. A CPU is saturated with only a few threads, whereas a GPU needs to have threads in the thousands before saturation occurs. This comes at a cost though; individual threads will invariably perform worse than the same one on a CPU. Therefore, the performance gained from the GPU is from executing highly parallelized code.

GPUs are efficient at solving data-parallel problems. Data-parallel means that a single task operates identically and independently on a set of data. In computer

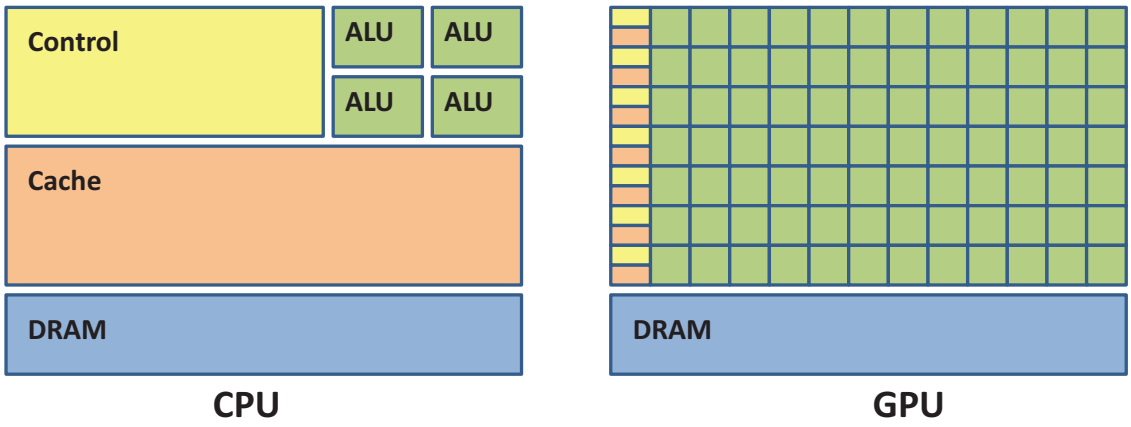

Figure 2.7: CPUs use large amounts of transistors for cache and control logic for a very limited number of threads. On the other hand, GPUs have very little thread overhead but have many ALUs for numerous threads. [13]

Algorithm 2.1: Calculate $\mathbf{c} = \mathbf{a} + \mathbf{b}$ on the CPU

```
1 for i ⇐ 1 to HEIGHT do
2 |   for j ⇐ 1 to WIDTH do
3 |   |   c_{i,j} = a_{i,j} + b_{i,j}
4 |   end
5 end
```

Algorithm 2.2: Calculate $\mathbf{c} = \mathbf{a} + \mathbf{b}$ on the GPU

```
1 i ⇐ current row
2 j ⇐ current column
3 c_{i,j} = a_{i,j} + b_{i,j}
```

architecture terms, it is equivalent to Single Instruction Multiple Data (SIMD) instructions. Figure 2.7 provides a graphical depiction of the difference between GPUs and CPUs.

To illustrate the difference between CPUs and GPUs, consider matrix addition. For a CPU the algorithm would look something like Algorithm 2.1. This algorithm processes each element one at a time in series. This can be a very slow process when the matrices become large. If the matrix has n elements, the algorithm will take n^2 iterations of the loop to complete.

On the other hand, the GPU code in Algorithm 2.2 creates one thread for each matrix element, adds that element from **a** and **b**, and then stores that value into the proper element in **c**. A GPU can execute hundreds of threads simultaneously. For example, if a GPU can process 1,000 threads at a time, it will take $n^2/1,000$ iterations to complete. In practice, speedups for GPU applications can be between 10 to 200 times faster than their CPU-only counterparts.

The remainder of this section will use NVIDIA CUDA terminology to explain the architecture of GPUs.

2.3.3 CUDA Terminology. When working with CUDA programs, it is necessary to understand some basic terminology. The *host* is the CPU, whereas the GPU is referred to as the *device*. A *kernel* is a small function that is executed on the device by a large number of threads.

Compute capability is a number assigned a GPU representing its computational capabilities. A higher compute capability number means the GPU can handle more advanced mathematical and programming operations. For example, compute capability 1.0, 1.1, and 1.2 can only handle single point precision operations, while the latest compute capability 1.3 has double precision. Also, 1.0 does not allow for any atomic memory operations, while 1.2 allows for shared and global atomic memory operations.

2.3.4 Thread Hierarchy. The thread hierarchy describes how the GPU executes threads as well as how threads interact with each other. When a kernel is sent to the device, it as assigned a *grid*. Grids are a logical mapping of the threads that are executed within the kernel. Each grid is made up us smaller components called *blocks*. These, in turn, are composed of individual threads. On the current generation of NVIDIA's GPUs, blocks can contain no more than 512 threads. Grid dimensions can be as large as $2^{16} \times 2^{16}$. Figure 2.8 shows the thread hierarchy.

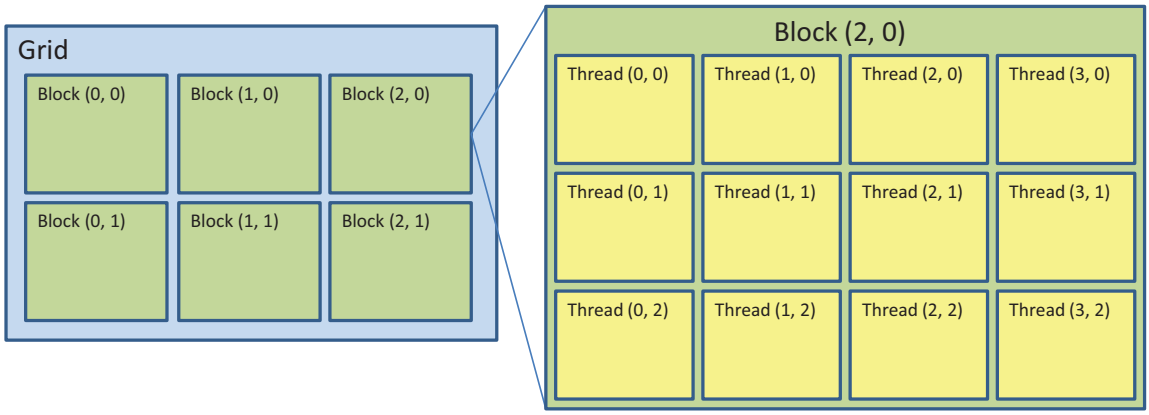

Figure 2.8: Every CUDA kernel is assigned a grid. Grids are subdivided into blocks. Each block contains up to 512 individual threads. Blocks and grids can be logically organized into multi-dimensional structures to simplify memory access calculations. [13]

Grids can be logically organized into a 1D or 2D layout, whereas a block can also be organized in a 3D layout. This organization can be useful to efficiently calculate the array index for each thread. For example, if the data processed by the kernel is a 2D image, then it would make sense to use a 2D grid and block structure. The current index for each thread can be calculated using built-in CUDA variables, as shown in Equation 2.12. This index can then be used within each thread to perform a certain operation on the image.

$$\begin{aligned} \texttt{column} &\Leftarrow \texttt{blockIdx}_x * \texttt{blockDim}_x + \texttt{threadIdx}_x \\ \texttt{row} &\Leftarrow \texttt{blockIdx}_y * \texttt{blockDim}_y + \texttt{threadIdx}_y \end{aligned} \tag{2.12}$$

where `blockIdx` is a vector representing the current location in the grid, `blockDim` specifies the width and height of each block, and `threadIdx` is a vector representing the current location in the block.

2.3.5 Memory Hierarchy. Memory hierarchy and memory access methods are important to understand to improve the speed of applications on GPUs. There are three different levels of memory: global, shared, and local. Two additional types of memory exist (texture and constant) but will not be covered.

Table 2.4: Coalescing memory writes is very important for GPU performance. This table summarizes the rules for memory accesses within a half-warp (16 threads) for devices of compute capability 1.2 or higher.

Byte Spread	Word Size
32 bytes	8-bit
64 bytes	16-bit
128 bytes	32-bit
128 bytes	64-bit

Global memory can be accessed by any thread, no matter what block it is in. This memory is the largest (most modern GPUs have more than 500 MB), but it is also the slowest. No memory reads from global memory are cached. In addition, since this memory is further away from the multiprocessor cores, there is a latency of about 400 to 600 clock cycles.

Shared memory is memory that can be accessed only by threads within the same block. Current GPUs have 16 KB of shared memory. Access to shared memory can be as fast as reading and writing to registers so long as there are no bank conflicts within a half-warp. A bank conflict occurs if two threads within a half-warp are reading or writing to the same 32-bit block of shared memory.

Local memory can only be accessed by an individual thread. The CUDA compiler places as many local variables as it can into registers, but if there is overflow, variables are stored in a reserved section of the GPU's global memory. This means variables should be chosen to fit in local registers or there will be a significant performance decrease.

The most important concept to understand about memory access is *coalescing*. This means that all half-warp memory accesses are within a given segment size of memory. Table 2.4 summarizes the rules for devices with compute capability 1.2 or higher. If these rules are followed, then only one memory access command is executed. If they are not followed, then individual memory fetch/store commands must be issued for each thread (16 individual memory fetch/store commands instead of one).

2.3.6 Synchronization. Synchronization is limited within the GPU. Only threads within the same block can be synchronized (using the `__syncthreads()` command). This limitation is because all the thread blocks are not executing at the same time. The GPU has a scheduler which submits blocks for execution on an individual Single Instruction Multiple Thread (SIMT) multiprocessor core. Each SIMT contains eight scalar processor cores. These cores are capable of executing one *warp* (or set of 32 threads) at a time. The NVIDIA GTX 280 has 30 SIMT miltiprocessor cores, which means it can execute only 240 blocks simultaneously. Synchronizing within a core (i.e., block) requires only four clock cycles, but synchronization between multiple blocks is much more costly because of the interprocessor communication.

2.3.7 Performance Notes. Writing efficient code on the GPU can be difficult. Memory bandwidth and latency are two important concepts that affect overall performance. As has already been mentioned, coalescing is also very important to speed up the execution of code on the GPU.

What is not as obvious is that sending data between the host and device can cause a bottleneck in a program. Communication between the CPU, GPU, and RAM is mediated by the northbridge, as shown in Figure 2.9. There is a limit of 12.8 GB/s between the CPU and northbridge. One-way communication between the CPU and GPU is limited to 8 GB/s with PCIe 16x Generation 2. An uncompressed, 24-bit, 1920x1080 (i.e., 1080p) image uses 5.9 MB of memory which would take about one millisecond to copy from the CPU to GPU. This means that about two milliseconds are used just to transfer data, without any useful computation.

Therefore, the amount of data-parallel operations executed on the GPU needs to be great enough such that their quick execution more than makes up for the transfer time to and from the GPU. For example, a single operation of matrix addition or transpose would probably execute more quickly on the CPU, while matrix multiplication would be quicker on the GPU.

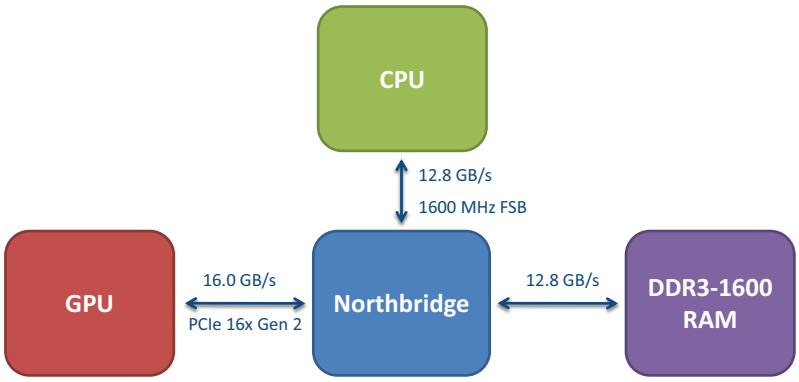

Figure 2.9: The CPU and GPU cannot communicate directly with each other. The northbridge mediates communication between the CPU, GPU, and RAM.

Device-to-device memory transfers refers to moving data to another portion of global memory on the same graphics card (i.e., device). These transfers have a bandwidth of up to 141.7 GB/s, which makes them not as susceptible to the bandwidth problems described above.

2.4 Literature Review

This section briefly presents related work that has been published in the literature. Most previous research focuses on image processing on the GPU. MTT has been well researched in single-core and cluster machines, but no research could be found that uses GPUs to improve MTT performance [14] [15] [16].

2.4.1 GPGPU Image Processing Libraries. In recent years, a number of libraries have been written for general purpose image processing on the GPU. The most well-developed one is called GPU Computer Vision (GpuCV), based off the more well known Open Computer Vision (OpenCV) library developed by Intel [17]. GpuCV offers a subset of OpenCV, and its data structures and functions are intended to be compatible with any software written with OpenCV. It includes functions for addition, multiplication, thresholding, color manipulation, Sobel edge detection, and discrete Fourier transforms. The GpuCV library is limited since it does not supply a GPU implementation of CCL or blob analysis, which are needed in MTT systems.

Algorithm 2.3: Parallel Implementation of CCL

input : i - row pixel that processor is to operate on
j - column pixel that processor is to operate on
globalFlag - flag memory location available to all processors
image - a binary image with blobs to be labeled
output: Image with uniquely labeled blobs

```
1  if image_{i,j} = 1 then
2  |  image_{i,j} ⇐ uniqueThreadID
3  |
4  |  repeat
5  |  |  globalFlag = 0
6  |  |  synchronizeAllThreads()
7  |  |  minimumLabel ⇐ minimum label of neighbor pixels
8  |  |
9  |  |  if minimumLabel < image_{i,j} then
10 |  |  |  image_{i,j} ⇐ minimumLabel
11 |  |  |  globalFlag ⇐ 1
12 |  |  end
13 |  |
14 |  until globalFlag = 0 ;
15 end
```

GPU programming, in particular CUDA, is emerging into other areas of industry as well. Medical applications are being explored. Research results indicate that a speedup of 13 times is achievable using CUDA for biomedical imaging [18]. Plugins have also been developed for commercial image manipulation applications, like Adobe Photoshop, to speed up image filters [19].

2.4.2 Parallel CCL Algorithms. Solving the CCL problem with parallel processors has been researched as early as 1983. Most methods that have been developed were written to only work on specific parallel architectures. The most common general parallel method is outlined in Algorithm 2.3 [20]. Essentially, each thread is assigned each pixel. If that pixel is part of a blob, it is assigned a unique label. Each thread then assigns itself to the minimum of its neighbors' labels until no reassignments have been made.

One problem with this algorithm is that it can take a long time to propagate a label if one of the blobs is irregular in nature (e.g., a snake shaped object wrapping around the image). This algorithm would be very inefficient in that situation, but it does perform well when all blobs are relatively small and non-convex.

More advanced parallel CCL algorithms are presented by [21], but they do not map well to the GPU architecture.

2.4.3 Fast Radial Blob Detector. The Fast Radial Blob Detector (FRBD), presented in [22], is an algorithm which finds the areas of a grayscale or color image with high gradients (or change in pixel intensity). These areas indicate the location of an edge or blob.

FFRB assigns one thread to each pixel. That thread calculates the average gradient between itself and its neighbors at a few predefined distances from itself. A threshold is then used to indicate whether a pixel is or is not part of an edge or blob.

Although some of this algorithm is implemented on a GPU, a significant portion of it is run on the CPU. This is because the GPU merely finds edges within the image, it does not apply a unique label to each of those edges. A minimum spanning tree is used on the CPU to combine all blobs and edges into clusters, calculate their centroid, and then pass that data onto the MTT algorithm.

2.5 Summary

The goal of this chapter was to provide the background necessary to understand the research presented in this thesis. Image processing techniques used to extract objects from a sequence of images were covered in Section 2.1. In Section 2.2 the fundamental principles of MTT were covered, from the sensor input to filtering and prediction. GPU programming concepts were introduced in Section 2.3. Some optimization techniques for the GPU were also briefly discussed. Finally, the chapter concluded with a brief overview of related research available in the literature.

III. Methodology

Image processing is a bottleneck in the performance of MTT. Many of the solutions to improve performance either are expensive or have very long development times. This research explores the use a low-cost and rapid-development hardware and software package to speed up image processing in MTT. This chapter describes the implementation of MTT software. There are three different versions of the software:

1. A complete self-contained MTT software package written MATLAB .
2. Slow image processing functions are converted to C code to improve overall MTT performance.
3. The same image processing functions are converted to run on the GPU. Two different versions of this software are developed:
   (a) No atomic operations available (Compute Capability 1.0)
   (b) Atomic operations available for shared and global memory Compute Capability 1.2+

3.1 MTT Software Development (MATLAB)

All of the algorithms used in the MTT software are described in detail in Chapter II. This section will briefly describe the MTT software sequentially and explain how it was implemented in MATLAB.

Figure 3.1 outlines the software functional flow. All functions in this MATLAB implementation are single threaded.

Optimizations are used for all of the different functions created for the MTT software. In MATLAB, this means `for` loops are only used as a last resort; instead, the code is vectorized. *Vectorized* code refers to using linear algebra functions on data instead of processing them one at a time in a loop. MATLAB is an interpreted language, which means it is very slow at executing loops, but it is optimized to execute linear algebra functions quickly.

3.1.1 Input Data. The MTT software is able to perform target tracking on any fixed frame-rate video data that can be read by the MATLAB function `mmreader`.

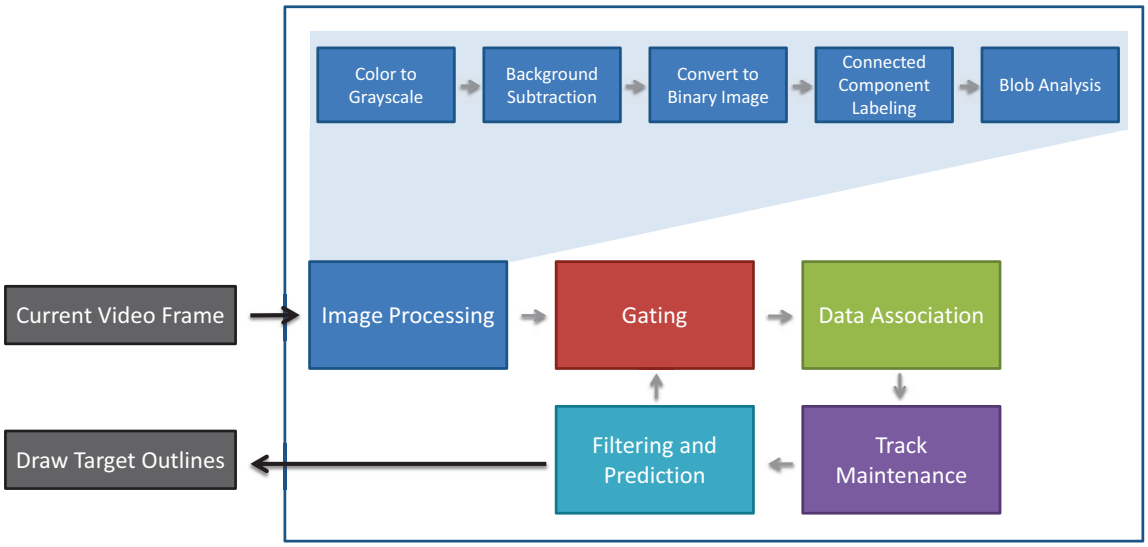

Figure 3.1: A detailed flowchart of how the MTT software processes each video frame.

On Windows, this includes AVI, MPEG-1, and Windows Media Video formats. The tracking software assumes that all pixels are updated each frame, which means only progressive-scan video should be used.

Real-time video capture functionality is possible in MATLAB with the Data Acquisition toolbox, but it is not implemented in the software since it does not add significant value to the research. The software processes the video feeds one frame at a time and does not look at future frames to track targets in the current frame.

There is one file that contains all the configuration variables for simulations. In this file, parameters can be set for the video, target model, Kalman filter, and track maintenance.

3.1.2 Output Data. In the configuration file, the user can specify the desired output. The most common output is displaying each video frame in real-time and outlining each target that is being tracked. Debugging data may also be printed out to the console along with a profile of how long each frame takes to process. Also, a detailed break-down of the time spent in each function can be generated.

Table 3.1: Summary of MATLAB image processing implementation. All functions were taken from the Image Processing Toolbox, except for a custom written threshold function.

Image Processing Block	MATLAB or Custom	Function Name
Color to Grayscale	MATLAB	`rgb2gray`
Background Subtraction	MATLAB	`imabsdiff`
Convert to Binary Image	Custom	N/A
Connected Component Labeling	MATLAB	`bwlabel`
Blob Analysis	MATLAB	`regionprops`

3.1.3 Image Processing. The image processing algorithms are covered in detail in Section 2.1. Table 3.1 summarizes how each of the image processing functions were implemented in MATLAB. As can be seen, almost all the functions were available in the Image Processing Toolbox. The "Convert to Binary Image" (or "Threshold") function was implemented in a single line of MATLAB code, as shown in Equation 2.3.

3.1.4 Gating. The measurement noise in the x and y directions are assumed to be the same. Using this assumption, the gates are drawn in a circle around the predicted target location. In Equation 3.1, the radius is calculated as a multiple of the covariance of the model's position state at the current time period.

$$r = \alpha\sqrt{P_{x_{1,1}}^2 + P_{y_{1,1}}^2} \tag{3.1}$$

where α is a predefined constant, P_x is the covariance matrix for the x-dimension KF, and P_y is the covariance matrix for the y-dimension KF. Note that the covariance matrix is not squared, rather it is the position state variable, a scalar value, that is squared.

If a track has not been initialized (i.e., only one observation has been made) then the gate is created by estimating the velocity of the target with a predefined constant. The gate is expanded each time frame until another observation is found or the track is deleted.

3.1.5 Data Association. GNN is the data association method used since it offers the highest accuracy for a single hypothesis target tracking system. Since writing an implementation of the algorithm is not important for this thesis research, a solution to the assignment problem was downloaded from The MathWorks, Inc. website [23].

3.1.6 Track Maintenance. Tracks have four distinct states: (1) *uninitialized*, (2) *initialized*, (3) *confirmed*, and (4) *deleted*. New tracks are made for every observation that has not been assigned to a previously created track. A new track begins in the *uninitialized* state.

Once a track is observed twice, its state and covariance matrices are initialized and KF propagate and update equations are executed each simulation time step (i.e., video camera frame rate). The track is then in the *initialized* state.

A track is *confirmed* after it is observed K_c times in the last M observations, both of which are constants defined in the configuration file. This is the "sliding window" method of track maintenance.

Finally, tracks are moved to the *deleted* state after they are not observed for the last K_d frames. Again, this is a constant defined in the configuration file. Once in this state, the KF and gate for the track are no longer used or updated.

3.1.7 Filtering and Prediction. The MTT software model used is the Singer model, which Section 2.2.5.2 describes in detail. This model was chosen because it is simple and will accurately track maneuvering targets. Filtering and prediction are done with a standard Kalman filter. All parameters for the KF and model are defined in the software configuration file. The observation matrix is $H = [1\ 0\ 0]$ since only the position of a target can be determined from an image.

3.1.8 Profile. A summary of the MTT software performance is provided in Table 3.2. This table was generated from processing 50 sequential frames in a fixed

Table 3.2: Summary of the time required by MATLAB to process one 1080p image in the MTT software. These samples were generated on a different machine than those on Table 4.2. The individual execution times are different, however the *proportion* (i.e., "% Time") of the execution times does not change significantly from machine to machine.

Function	**Time (ms)**	**% Time**
Read Current Video Frame	140.0	34.2 %
Color to Grayscale	71.9	61.2 %
Background Subtraction	6.2	
Convert to Binary Image	2.8	
Connected Component Labeling	29.6	
Blob Analysis	140.1	
Gating	3.3	4.5 %
Data Association	0.9	
Track Maintenance	0.0	
Filtering and Prediction	14.4	
Total:	409.2	

frame rate, 1080p AVI video file encoded with the Indeo® 5.10 video codec. The two bottlenecks for performance are reading the video frames into memory and performing image processing.

The processing times for many of the functions listed in Table 3.2 are data-dependent. In particular, the sample data used had a maximum of three targets moving at a time. For larger scenarios (e.g., 100+ targets) the times will increase significantly for target tracking, especially for data association. This increase is bounded by an exponential expression of the number of targets and observations.

3.2 C MEX Implementation

The MATLAB implementation can only process HD images at about two frames per second (FPS). Nearly a third of that time is dedicated to reading in the video frames from a file. Image processing takes nearly two-thirds of the time, which means it is a good candidate to port into C MEX code.

Algorithm 3.1: Generalized structure of MATLAB MEX files

1 Read input variables from MATLAB
2 Perform desired function
3 Store output variables in MATLAB format

3.2.1 Overview of MEX Files. MEX files allow a programmer to convert MATLAB functions into C or FORTRAN code to speed up their performance. MATLAB performs well with vectorized math, but it is very slow at executing loops, which are frequently used in image processing. The general format of MEX files is shown in Algorithm 3.1. Like the MATLAB implementation, all the functions for the C MEX implementation are single threaded.

3.2.2 Image Processing. The C MEX image processing functions are implemented as described in Section 2.1. For the first three functions, this is a matter of a single equation within a `for` loop that iterates over each pixel. The CCL and blob analysis functions are a bit more complicated, but the code is written to reflect the algorithms and principles described in Section 2.1.

3.2.2.1 Color to Binary Image. Converting the image processing functions into C code is a straight-forward process. Equations 2.1, 2.2, and 2.3 describe the behavior for the MEX program to operate on each pixel for the first three stages of image processing. These stages are shown as a flow chart in Figure 3.1.

3.2.2.2 Connected Component Labeling. The two-pass CCL method, depicted in Algorithm 3.2, and blob analysis are implemented as described in Section 2.1. Custom-built linked-list and union-find array structures were used to store the CCL equivalence table.

The two-pass algorithm uses two `for` loops to iterate through the image. On the first loop, each non-background pixel is assigned the minimum label of its neighbors. The neighbor labels are then stored in an equivalence list to indicate that the labels

are synonymous. If there are no neighbor labels, then the pixel is labeled with a unique number.

On the second pass, each pixel is assigned to the value of its equivalence class. For example, if there are three different equivalence sets (i.e., three blobs),

$$\mathbf{1} = \{2, 4\}$$
$$\mathbf{2} = \{1, 3, 5\}$$
$$\mathbf{3} = \{6\}$$

any pixel that contains a label within set **1** (i.e., 2 or 4) will be assigned the label **1**. The same applies for the other sets. At the end, each blob is assigned a single unique label for all of its pixels.

3.2.2.3 Blob Analysis. The blob analysis function in MATLAB is nearly identical to the implementation described in Section 2.1. The tables used to generate the x- and y-coordinates are stored as 2D array structures. Once these arrays are populated, the area and coordinates for the blobs are calculated within two `for` loops that are used to add up their contents.

3.2.3 Profile. A timing profile (the same setup previously used) tests the speed of the image processing C MEX implementation. Table 3.3 summarizes the results. Overall, the C MEX implementation is about 85% faster than the MATLAB Image Processing Toolbox.

As seen in Table 3.3, some functions are faster with the custom MEX implementation, and some are significantly slower. One would expect a well-written MEX file to run quicker than pure MATLAB code since it is pre-compiled. There are a variety of reasons why the custom MEX implementation may be slower:

1. MATLAB now uses a Just-in-Time (JIT) compiler to compile loops and other frequently run code (similar to .NET and Java).

Algorithm 3.2: The algorithm to solve CCL with a serial two-pass method.

```
1  // First Pass
2  maxLabel ← 2
3  for i ← 1 to NUM_ROWS do
4  |  for j ← 1 to NUM_COLUMNS do
5  |  |  if image_{i,j} ≠ BACKGROUND then
6  |  |  |  if Labeled Neighbors Exist then
7  |  |  |  |  image_{i,j} ← minimum label of neighbors
8  |  |  |  |  union neighbor labels
9  |  |  |  else
10 |  |  |  |  image_{i,j} ← maxLabel
11 |  |  |  |  maxLabel ← maxLabel + 1
12 |  |  |  |  create equivalence entry for image_{i,j}
13 |  |  |  end
14 |  |  end
15 |  end
16 end
17
18 // Second Pass
19 for i ← 1 to NUM_ROWS do
20 |  for j ← 1 to NUM_COLUMNS do
21 |  |  image_{i,j} ← find(image_{i,j})
22 |  end
23 end
```

Table 3.3: Summary of the time required by MATLAB C MEX functions to process one 1080p image in the MTT software. Overall, the MEX implementation is about 85% faster than the MATLAB Image Processing Toolbox implementation.

Function	**MATLAB Time (ms)**	**C MEX Time (ms)**	**% Speedup**
Color to Grayscale	71.9	38.1	+88.7
Background Subtraction	6.2	13.5	−54.1
Convert to Binary Image	2.8	2.5	+12.0
Connected Component Labeling	29.6	57.8	−48.8
Blob Analysis	140.1	23.3	+501.3
Total:	250.6	135.2	+85.4

2. MATLAB has the ability to compute with multiple threads across processor cores.
3. Some MATLAB functions have their own MEX files for computationally intensive operations that cannot execute quickly in native MATLAB code.

In this case, it turns out all the functions that are slower with a custom MEX file are in fact already implemented as MEX files in the toolbox, the source code of which is not publicly available. The converse is also true, all the faster functions are implemented as pure MATLAB in the toolbox.

3.3 GPU Implementation

The final, and most important, step of the research is to port some of the MTT code from MATLAB onto the GPU. CUDA is the programming language used to port the functions to execute on a NVIDIA graphics processor. Because of limitations of the GPU, only certain functions can be efficiently be implemented with CUDA. In this software, most of the image processing functions are data-parallel algorithms that are well-suited to GPU implementation. There are two different types of CUDA code written for the MTT software, one uses atomic operations and one does not.

3.3.1 Implementable Functions. Before writing any code to port functions, a brief analysis of the problem structure must be done in order to determine if it is feasible to implement on a GPU. As discussed in Section 2.3, good candidates are functions that are highly data-parallel. Ideally, the problem can be solved by thousands of threads simultaneously rather than being limited to just a few.

As discussed in the previous section, two-thirds of the MATLAB MTT software code is spent on image processing. Many image processing functions are inherently data-parallel since the same operation is independently repeated on every pixel. This means that it is reasonable to attempt to try to speed up at least some of the image processing functions by implementing them on the GPU with CUDA.

3.3.2 Performance Considerations. The bottleneck with GPU computations is memory bandwidth. It takes about a millisecond to transfer a 24-bit, 1080p image between the CPU and GPU. If the entire image is transfered between the CPU and GPU after each image processing stage, then the overall performance will be de-

graded. A better solution is to transfer the current frame and background once at the beginning, perform the image processing computations on them, and then transfer back the blob analysis data (list of centroids and area for all the blobs in the image).

Performance can also be decreased if global memory is accessed too much or in the wrong way. A global memory read or write operation takes between 400 to 600 clock cycles. The amount of memory transfers can be decreased by caching data (whenever possible) into local and shared memory for blocks and threads to access during computation. Afterwards, the data can be stored back in the global memory. Also, reads and writes to global memory need to be coalesced. Non-coalesced operations require separate memory operations for each thread, while coalesced actions are able to be executed in one operation per half-warp. Following these rules can significantly increase the performance of the functions implemented on the GPU.

3.3.3 Compute Capability. Each NVIDIA GPU is assigned a compute capability level that describes its mathematical and programming capabilities. Table 3.4 provides a brief summary of the differences between compute capabilities. Because of these different levels, there were two different types of GPU implementations written for the MTT software. The first implementation works on any compute capability level since it does not use atomic memory operations. The second implementation does use atomic memory operations and therefore can only be executed on GPUs with compute capability 1.2 or greater.

Only the blob analysis function is different between the two implementations. All other functions are identical. The unique structure of the blob analysis problem made it more efficient and robust to solve with atomic operations.

3.3.4 Color to Binary Image. This section describes the CUDA implementation of the image processing functions "Color to Grayscale" (`rgb2gray`), "Background Subtraction" (`imabsdiff`), and "Convert to Binary Image" (`threshold`). All three of these functions fit perfectly into the data-parallel model for stream processing. Each

Table 3.4: A brief comparison between different compute capabilities on NVIDIA GPUs. Precision refers to the maximum variable precision the ALU hardware can natively handle.

	Atomic Memory		**Precision**	
Compute Capability	**Shared**	**Global**	**Single**	**Double**
1.0			X	
1.1		X	X	
1.2	X	X	X	
1.3	X	X	X	X

function performs an identical operation on every pixel of the image and there is no interdependence between pixels. Therefore, each pixel can be assigned to a single thread to perform the computations. In a 1080p image, there are 2,073,600 pixels. The block size is set to the maximum number of threads (512) as to maximize the number of threads being simultaneously executed on the GPU. The kernel bodies for each function are described in Equations 2.1, 2.2, and 2.3.

Originally, all three functions were executed as three separate kernels. However, in the final product they are combined into one kernel to improve their overall performance. Instead of between two to five global memory read and write operations per pixel per function, there are only four reads/writes total (8-bit RGB values, 8-bit background pixel intensity, threshold value, and binary pixel value). The input global memory variables are stored in local memory, processed through the three functions, and then the binary image is written back to global memory. All memory operations are coalesced on NVIDIA devices with Compute Capability 1.2 or above.

3.3.5 Connected Component Labeling. As discussed in Section 2.4, solving CCL on parallel architectures is a well known problem, but no implementation has been published to date that works efficiently on a GPU. The final algorithm used is very similar to the one presented in [20], however it was slightly modified to work correctly on the GPU hardware.

Algorithm 3.3: Parallel Implementation of CCL on CUDA

input : *image* - a binary image with blobs to be labeled
output: *image* - original image with uniquely labeled blobs

```
1  start kernel_1: (x, y are pixel coordinates calculated by each thread)
2  if image_{x,y} ≠ BACKGROUND then
3  |  image_{x,y} ← (blockIdx_x * blockDim_x + threadIdx_x + 1)
4  end
5  end kernel_1
6
7  repeat
8  |  start kernel_2:
9  |  if image_{x,y} ≠ BACKGROUND then
10 |  |  image_{x,y} ← minimum label of neighbors
11 |  end
12 |  end kernel_2
13 until kernel_2 has not modified any pixels ;
```

3.3.5.1 Algorithm Implementation. Algorithm 3.3 is a pseudocode representation of how CCL is solved using CUDA. A few optimizations are implemented to make it run more quickly. Since each time a kernel is invoked from the CPU takes some overhead to execute, the $kernel_2$ contains a loop (executed a constant number of times) that relabels each pixel the minimum of its neighbors. This loop is unrolled to minimize the number of branch mispredictions. Global memory accesses are coalesced (whenever possible), and variables are cached in local memory to prevent too many global memory accesses.

3.3.5.2 Label Minimization. While Algorithm 3.3 does technically solve the CCL problem, it is not sufficient for an image processing application. This is because the labels it produces are neither sequential nor minimal. For example, an image with three blobs may generate labels 63984, 12, and 345. The blob analysis algorithm requires that they be labeled minimally from 1 to n, where n is the number of blobs in the image.

Algorithm 3.4: CCL Label Reduction

1 Remove non-label pixels from image
2 Delete sequential duplicates in the 1D image array
3 Sort array with radix algorithm
4 Delete sequential duplicates
5 Store reduced labels in $O(1)$ lookup structure
6 Relabel original image

This is non-trivial to solve on the GPU with CUDA. The GPU is able to solve many identical and *independent* operations at the same time. However, reducing the labels creates a data dependency between the pixels in the different blobs.

In order to solve this problem, a new six-step algorithm (created for this research) is used to efficiently minimize the labels on the GPU. Algorithm 3.4 provides a brief summary of the steps. The solution used requires six sequential kernels described in detail in the following paragraphs.

The number of pixels must be reduced to make the sorting more tractable in Step 3. To do this, first all non-labeled pixels are removed from the image, and the remaining pixels are stored in a 1D array. Next, the sequential duplicate elements in the 1D array are removed. This does not globally remove duplicates, only duplicates that are immediately next to each other in the array.

After the number of pixels have been reduced, the array can be sorted. This is done with the radix sort since it can be efficiently implemented on the GPU with CUDA. Then all the duplicate elements in the array are removed. Since the array is sorted, this is the same as Step 2.

A large array is created where the reduced label is stored in the index of its original label. For example, if the label was originally 8347 and it was reduced to 5, then it would be stored in array *minLabel* as $minLabel[8347] \leftarrow 5$. This allows the relabel lookup to be executed in $O(1)$ time.

Finally, the original image is relabeled with the reduced labels. At this point, each blob is labeled with a unique and minimized value. Therefore, the CCL algorithm is complete.

This algorithm performs well on data sets where there are a large number of background elements (Step 1 and 3) with wide or tall blobs (Step 2). Fortunately, in many target tracking applications, the density of targets in images is relatively low. Reducing the number of elements sorted by Step 3 is essential to keep this algorithm efficient because sorting is, at best, an $O(n \cdot log(n))$ operation.

3.3.5.3 Unsuccessful Optimizations. There were a number of optimizations that were attempted, but did not significantly improve the performance of CCL. First, 2D array structures were used to simplify the calculation of the current row and column for each thread's pixel. This prevents using two computationally costly division operations. Also, the blocks were arranged in a 16x16 structure with all the neighboring elements cached in shared memory. This significantly reduced the number of global memory accesses for each block.

The 16×16 block was solved by propagating minimum labels until no pixels were changed. Moving the loop into the kernel decreased the number of times the kernel was called by the CPU. The image was processed by these kernels until no more changes were made in any pixel's label.

3.3.6 Blob Analysis. Two different implementations of blob analysis are implemented to work with different types of graphics cards, with and without atomic operations. Compute capability 1.0 does not have any atomic operations, while compute capability 1.2+ has atomic operations for shared and global memory.

An atomic operation guarantees a thread that it will be able to complete a sequence of commands on a memory location without any other thread interrupting it. For example, if thread a attempts to increment the memory location x, it will first read the contents of x, increment it in local memory, then store it back to x. Without

atomic operations, it is possible for thread b to write a different value to x after a has read the previous value, but before it has stored the incremented value. Then the value that a writes to x is incorrect. If atomic operations are supported, then thread a can guarantee that it will be able to properly increment the value in x.

Both implementations use seven separate kernels to calculate the area and centroid of each blob. Two kernels are used to generate tables that specify the number of pixels for each blob in each row and column.

Next, one kernel is executed that calculates the area of each blob. It does this by summing up the number of pixels in each blob from the previous table generated for blob row pixel count.

Two more kernels are used to weight the pixel counts in the tables generated in the first two stages. The pixel count is multiplied by the column or row location. For example, if blob a has x pixels in row y, then it is assigned the weighted value $x \cdot y$.

Finally, two more kernels calculate the x and y coordinates of the centroid. This is done by summing up the weighted sums for each column and row, and dividing that sum by the area of the blob.

3.3.6.1 No Atomic Operations. The no-atomic-operation implementation of the first two kernels is complicated. One block is assigned to each row and column of the image. The block first creates a shared array (initialized to zero) where each element x represents the number of pixels of blob x that are in that block's row or column. Next, the pixel values for that row or column are loaded (using coalesced memory operations) into shared memory (i.e., cached). Each thread in the block is assigned to one blob. The thread then loops through each pixel in that row or column and counts the number of times it sees a pixel from its blob. After all the threads have completed, the shared array is stored back into global memory.

3.3.6.2 Atomic Operations. In the atomic-operation implementation, the kernels are much simpler. Each pixel is assigned to one thread. That thread

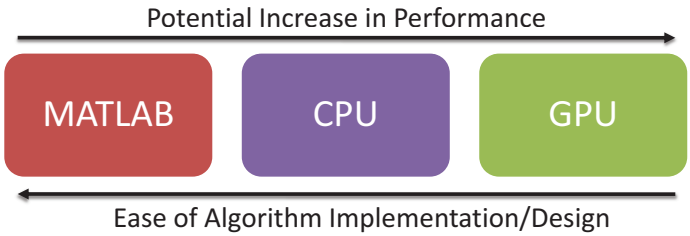

Figure 3.2: A brief visual summary of the difficulty of implementation compared to potential performance increase.

simply updates the row and column table global memory location for the blob label of its pixel.

3.4 Summary

This chapter briefly covered the capabilities of the MTT software, the steps used to implement it, as well as the algorithms used for each stage. The MATLAB and C MEX implementations are meant to be used as baselines to be compared against the GPU implementation. The primary performance metric used is execution time (lower being better). These execution times can be compared to a baseline to provide the overall percentage speedup of different implementations. For instance, if the MATLAB implementation of a function takes 100 milliseconds to complete and the MEX version completes in 50 milliseconds, then the MEX function is 100% faster than the MATLAB function.

Figure 3.2 provides a brief summary of the difficulty of programming versus the potential performance increase for each of the three different implementations. The GPU implementation has the biggest potential increase in performance, but it is also more difficult to develop and implement algorithms.

IV. Results

After developing the MTT software, it is necessary to test its functionality. There are three main components of the MTT software: (1) read in current video frame, (2) image processing, and (3) target tracking algorithms. This research effort concentrated on porting the image processing functions to execute on the GPU. Follow-on research efforts can investigate porting the other two components of the MTT software. This chapter summarizes the results found from researching the use of GPUs to augment the MTT software.

4.1 Limitations

For this investigation, a limited number of functions were converted to execute on the GPU. Therefore, choosing which functions were converted was accomplished by using two criteria. First, the function had to pose a significant bottleneck to the MTT software performance. Second, a preliminary analysis of the function needed to indicate that the performance could be improved when ported onto the GPU.

A breakdown of the time spent executing each function facilitates the process of choosing which functions are converted to the GPU. Table 3.2 provides the exact times, but they are summarized visually in Figure 4.1. Applying Amdahl's Law helps choose what order to optimize the functions (in order): (1) image processing, (2) read current video data, and (3) tracking algorithms [24].

4.2 Functions Not Implemented on the GPU

Reading current video data and tracking algorithm functions are not implemented as GPU functions in the MTT software. This section briefly explains why they are not and how they would probably perform if they were.

4.2.1 Read Video Data. The MTT software spends over a third of the time just reading video data for the video frames. This is because of the time-intensive steps involved with reading in each frame. MATLAB has to request a file with a

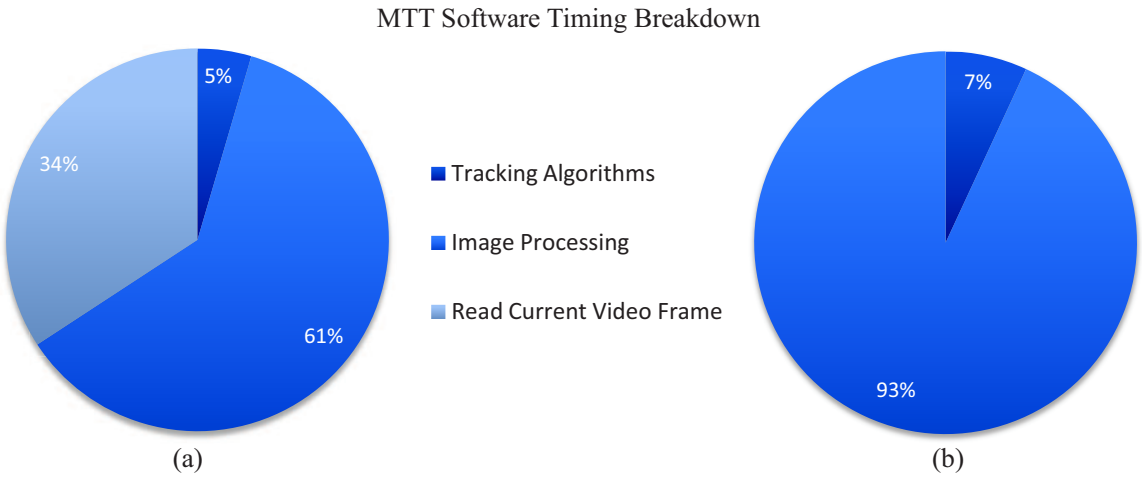

Figure 4.1: These pie charts represent proportion of time used for different types of functions. In (a) the percentages represent what is actually observed in the baseline MATLAB software, whereas (b) shows what the breakdown is without reading in the current video frame.

system call. The operating system then accesses the hard disk and finds the different fragments of the video frame. To add to the problem, the video is not cached in local memory and large video files are fragmented across the disk. Finally, the video is decoded and placed in a MATLAB data structure.

To speed up the performance of the MTT software, the video data could be prefetched or custom hardware could be used. Prefetching the video data would add a time delay to the real-time MTT software. In a real-time system the video data has to be processed as quickly as it is received from the imaging device. If the data is prefetched, then there would be a one-frame delay between what the camera is viewing and what is produced at the output of the MTT software.

Custom hardware would be a better option because it could potentially decrease the execution time by communicating on a low-latency, high-bandwidth data bus. However, such custom hardware would significantly increase the cost of the system. If the imaging sensor sends the uncompressed image data directly to low-latency memory (e.g., RAM), then the CPU would no longer have to access the hard drive or decode the video. In an embedded system, the video camera may continuously feed

the image data to the CPU or GPU and it could be processed in real-time. Any of these methods would significantly decrease the time spent reading the video data.

4.2.2 Tracking Algorithms. Figure 4.1(b) shows what the MTT software timing breakdown would be if there was no delay in receiving video data. With this data set, target tracking algorithms only use 7% of the execution time, compared to the 93% spent on image processing. Since this is such a small percentage, the time was better spent converting image processing functions onto the GPU rather than target tracking functions.

The main variable that changes the execution time for target tracking is the number of detected targets. Increasing the number of targets increases the time it takes to track them. For some functions this is a linear increase, however gating and data association increase more rapidly. The general rule of thumb is that the tracking functions may run quicker on the GPU for a large number of targets (1,000+), but for smaller numbers, the CPU is probably the better choice.

4.2.2.1 Gating. Gating is a $O(n^2)$ operation since the distance between every target must be measured with respect to every every target. Since these measurements are independent, they can be spread across multiple cores on the GPU to process more efficiently.

4.2.2.2 Data Association. The computational complexity of data association depends on what type of implementation is selected. For a simple algorithm choice, like NN or greedy, it can be completed in linear time. However, more complicated algorithms, like GNN or multiple hypothesis tracking (MHT), are nonlinear.

NN fits into the data-parallel paradigm since all targets are assigned the observation that is closest. The greedy algorithm also can be implemented on the GPU by using atomic operations to remove observations when they have been assigned a target. However, a more detailed analysis must be done to determine if more advanced algorithms would efficiently execute on the GPU.

4.2.2.3 Track Maintenance and Filtering/Prediction. Track maintenance and filtering/prediction are the two tracking stages that increase linearly with respect to the number of targets. This is because the same constant-time operation is executed on each target, regardless of how many other targets there are. For a large number of targets (1,000+), it would be reasonable to implement these algorithms on the GPU. However, for a smaller numbers of targets, these functions will execute more quickly on the CPU.

4.3 Experimental Setup

All of the test setups for the software have a number of similarities. The video data is real (not simulated) footage taken by a high-definition camera. The footage is shot at 30 frames per second, 1920×1080 pixels, and uses progressive scan (entire image is updated each frame).

The experiments use a simple MTT setup. The targets are one inch in diameter, semi-transparent rubber balls of various colors (green, blue, yellow, and purple). Since the image processing for this software uses background subtraction, it tracks targets best when there is a high contrast between the background and the targets. For this reason, the background used is a concrete sidewalk and the footage is taken at noon on a cloudy day to minimize the shadows. This is a best-case scenario for the MTT software, however the main purpose of these tests is to determine the *speed* of the *image processing* for MTT. The software has difficulty detecting valid targets in low-contrast scenarios since it uses a simple form of background subtraction. Again, this is not relevant to the *speed* of the image processing functions.

After a variety of simple test runs, it was determined that a threshold value of roughly 0.1 works best for the binary image step of the image processing, as described in Section 2.1. Recall from Equation 2.3, this means any pixel intensity above $\lfloor 0.1 \cdot 255 \rfloor$, or 25, will be stored as a foreground pixel (white), and any pixel intensity less than 25 will be stored as a background pixel (black). In a real application, this threshold value would most likely be automatically determined by analyz-

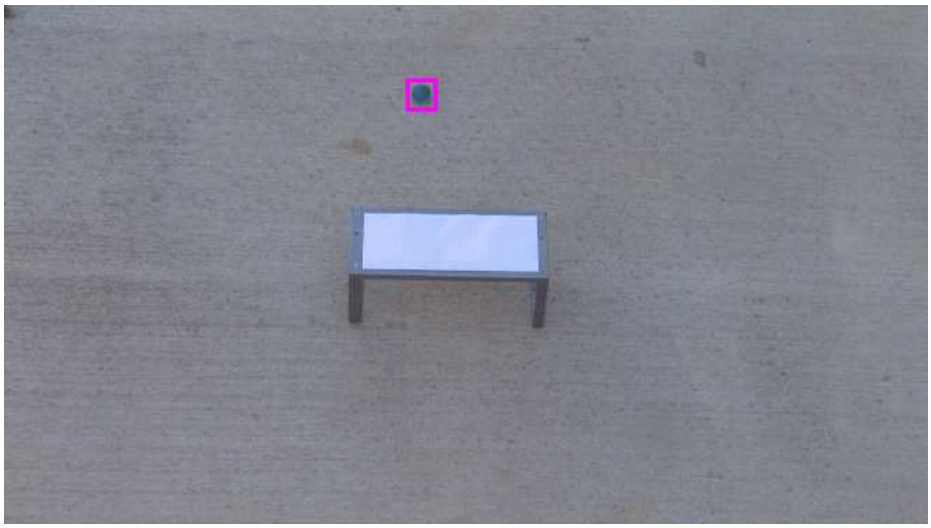

Figure 4.2: A single sample frame from Scenario 1. The blue target is obscured by the bridge for a portion of its movement from the top to the bottom of the frame.

ing a sequence of images. The Image Processing Toolbox in MATLAB contains the `graythresh` function which is useful in calculating a threshold value for a grayscale image. It minimizes the overall variance between the resulting black and white pixels by using Otsu's method [25].

4.4 Scenario Descriptions

Three different scenarios are used to test the functionality of the MTT software and to determine the performance difference between the different implementations (MATLAB, C MEX, and GPU). As already mentioned, the film is taken at noon to minimize the shadows and maximize the contrast between the targets and the background. Each scenario is selected to test a different aspect of image processing for MTT. They test performance of tracking: (1) a single target, (2) multiple targets with good contrast, and (3) multiple targets with more challenging contrast.

4.4.1 Scenario 1 - Single Target. The purpose of this scenario is to determine if the software is capable of detecting a single moving object. It also tests the software's ability to track an object even after is has been obscured in some way (e.g., moves under a bridge). Figure 4.2 shows a single frame of the scenario. A single blue

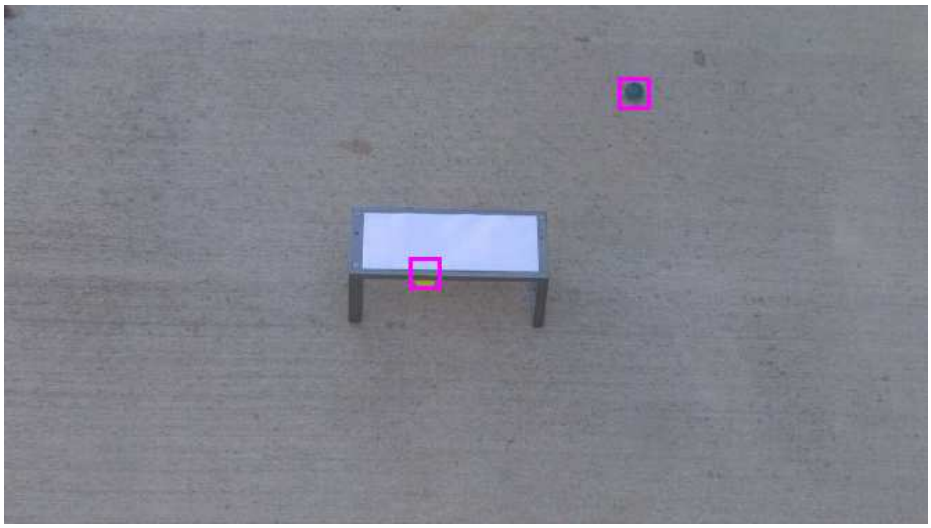

Figure 4.3: A single sample frame from Scenario 2. There are two targets in the frame. The yellow one is partially blocked by the bridge in the middle of the frame, while the blue one is completely visible. Both targets move from the top to the bottom of the frame. Only the yellow one moves under the bridge.

target moves into the scene from the top and exits at the bottom. In the middle of the scene, the target moves under a "bridge" and is hidden from view for a few frames.

4.4.2 Scenario 2 - Multiple Targets, Good Contrast. In the second scenario, the ability to track multiple targets is tested. Two targets move across the scene, and one is blocked by an obstruction for a few frames. Figure 4.3 is a snapshot of one frame from this scenario.

4.4.3 Scenario 3 - Multiple Targets, Poor Contrast. In the third scenario, five targets are in the scene. Four of them move under the bridge in the middle of the scene, while the fifth does not (Figure 4.4). The targets move from the right edge of the scene to the left. This scenario tests the ability of the MTT software to track a more complicated scenario. Also, since there are more blobs, the image processing portion of the software has more objects to detect and therefore may potentially run slower.

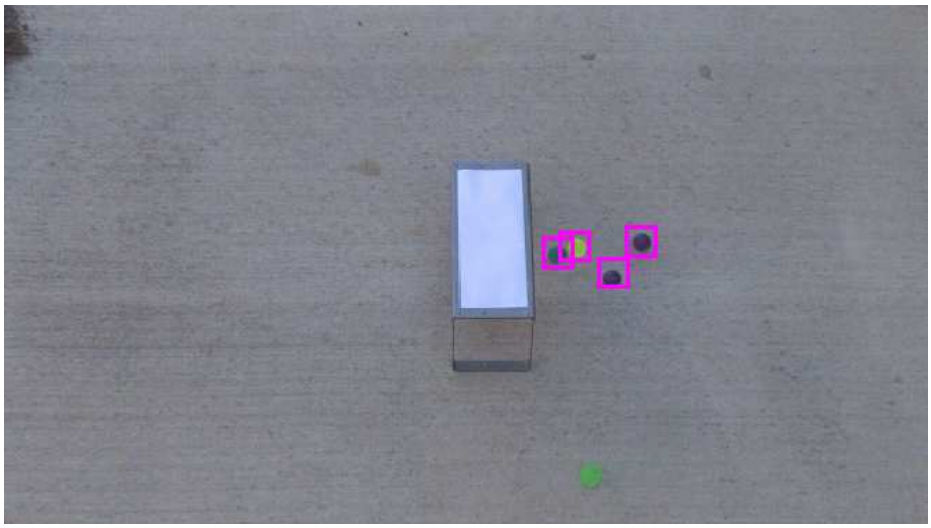

Figure 4.4: A single sample frame from Scenario 3. There are five targets in the scene. Four of them are detected by the software, the green one is not because of the low contrast between it and the background.

4.4.4 Threshold Value for Scenarios. A final note should be made about choosing or calculating the threshold value. If the value chosen is too large, then some or all targets will not be detected by the MTT software (as with the green target in Scenario 3). This decreases the number of false targets, but it also decreases the probability of detection. On the other hand, if the threshold value is too low, then too many false targets are detected even though most real targets are detected. Having too many targets can significantly decrease the speed of the image processing. In particular, a high number of false targets will exponentially slow down target tracking because of the assignment problem in GNN, as described in Section 2.2.

4.5 Results

This section details how each of the different implementations of the MTT software perform on the different scenarios. The four different implementations are:

1. Pure MATLAB and Image Processing Toolbox implementation
2. Image processing functions written as C-MEX code
3. Image processing functions written for NVIDIA GPU
   (a) Compute Capability 1.0

Table 4.1: A brief summary of the main components of the hardware used to test the software.

Component	**Description**
Model	Dell Precision T7400
Operating System	Windows Vista Enterprise 32-bit
Processor	Intel Xeon CPU X5482 @ 3.20 GHz (2 quad-core processors)
Memory	4×1GB 800 MHz DDR
Graphics Card	NVIDIA GeForce GTX 280 1.3 GHz Processor Clock 1GB Memory 240 Processor Cores

(b) Compute Capability 1.2+

The metric for performance of the image processing functions is the CPU time needed to complete each image processing function. A lower execution time means a better performance. The CPU time is calculated using the MATLAB `profile` function and CUDA timing functions. The average frame processing time for each function on each scenario is used to compare performance. Table 4.1 outlines the computer hardware and software specifications used to test the MTT software.

The test results are summarized in Table 4.2 and Figure 4.5. Table 4.2 is organized into the four different implementation types and the performance results for each of the three scenarios. The average of the three scenarios is taken to provide one performance number for each implementation. The final GPU implementation is 2.87 times faster than the original MATLAB implementation. That is an improvement from about 8 FPS to 23 FPS.

Figure 4.5 provides a brief summary of the performance results. Parts (a) - (c) show the execution time for the image processing functions. Part (a) shows the execution time for the first three functions (color to grayscale, background subtraction, and convert to binary image). These three functions are combined into one as an optimization on the GPU. Part (d) shows the overall execution time of all of the image processing functions.

Table 4.2: A summary of all the results from the scenarios with the four different implementations. The three columns under each implementation represent the three different scenarios used to test the software. The final implementation is 2.87 times faster than the baseline implementation.

Function	**Time (ms)**											
	MATLAB			**C MEX**			**GPU 1.0**			**GPU 1.2+**		
	S1	**S2**	**S3**	**S1**	**S2**	**S3**	**S1**	**S2**	**S3**	**S1**	**S2**	**S3**
Overhead	n/a	n/a	n/a	n/a	n/a	n/a	6.12	6.10	6.15	6.23	6.16	6.24
Color to Grayscale	46.54	45.95	45.59	30.28	27.67	28.46						
Subtract Background	1.77	1.68	1.95	4.11	4.70	5.25	2.45	2.44	2.48	2.88	2.89	2.92
Threshold	3.23	2.80	3.46	2.34	3.63	3.69						
CCL	5.26	4.96	6.15	38.55	33.48	38.12	29.24	17.28	30.34	29.41	17.41	30.70
Blob Analysis	72.81	65.73	69.37	9.47	8.64	11.80	79.12	67.97	77.27	9.10	8.44	9.27
Total Time:	129.61	121.12	126.52	84.75	78.12	87.32	116.93	93.79	116.24	47.62	34.90	49.13
Average Time:		125.75			83.40			108.99			43.88	
Average FPS:		7.95			11.99			9.18			22.79	
Average Speedup:		1.00			1.51			1.15			2.87	

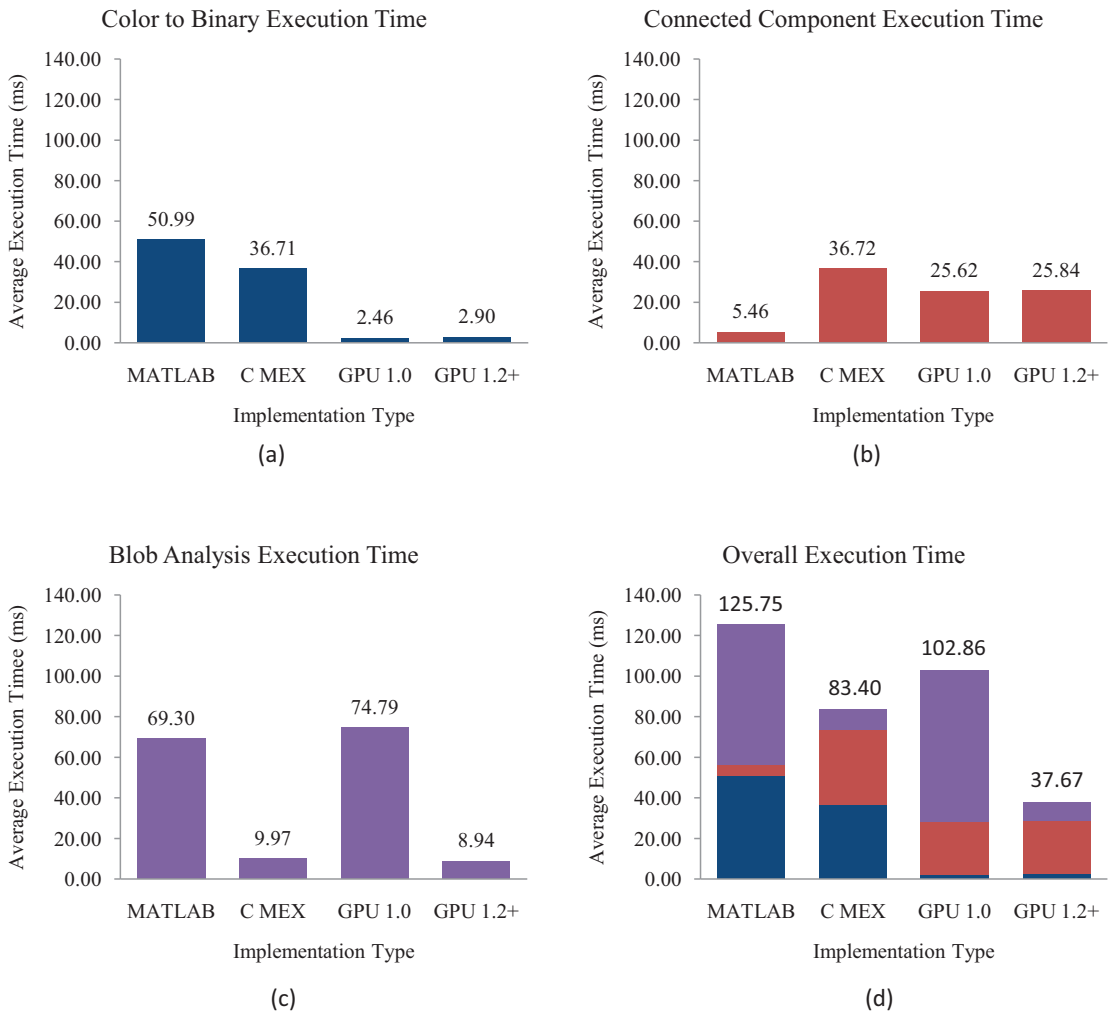

Figure 4.5: These bar charts represent the average execution time for different image processing functions. The pure MATLAB implementation is used as the baseline. Part (d) shows the overall execution time for the entire image processing.

4.6 Interpretation of Results

As expected, the final implementation is significantly faster than the original MATLAB implementation. However, there are a few unexpected results. The compute capability 1.0 implementation is only slightly faster than the MATLAB implementation, and actually slower than the C MEX implementation. Table 4.2 provides some insight as to why this is the case. The blob analysis function takes nearly 80 milliseconds to complete on the GPU, whereas it finishes in about 10 milliseconds on

the CPU. This is due to the complexity and inefficiency of the GPU 1.0 implementation, as described in Section 3.3.6. By adding the ability to use atomic operations, the blob analysis performance improves significantly. The GPU 1.2+ implementation is about 8 times faster than the MATLAB implementation, whereas the C MEX implementation is about 7 times faster and the GPU 1.0 implementation is 0.93 times as fast.

The C MEX implementation of blob analysis is significantly faster than the MATLAB version. This is due to the poor coding of the `regionprops` function in the Image Processing Toolbox. After analyzing `regionprops` with MATLAB's `profile` tool, it was found that most of the time is spent passing and parsing arguments rather than actually calculating the blob attributes. This problem is supposed to be fixed in the MATLAB 2009a release.

All of the custom-written CCL functions are slower than the original MATLAB Image Processing Toolbox versions. On the C MEX implementation, this can be attributed to the inefficient use of random memory on the heap. On the GPU, the slow down is largely because of global memory access, blob size, and coalesced reads. The CCL algorithm assigns one thread to each pixel. The threads continue to assign their pixel to the smallest label of its neighbors until all pixels are assigned to the local minimum. Each thread reading and writing to global memory takes up to 600 clock cycles. With the large size of the blobs in the scenarios (around 1000 pixels), there are a number of global memory operations. On top of that, the minimum label is always in the upper-left corner of the blob, which takes longer to propagate to all of the pixels in the blob than if it were in the middle. Also, the memory reads and writes are not always coalesced because of the branching logic in the CCL kernel.

The best performance improvements are seen in converting the color image to a binary image (first three steps in Figure 2.1). The GPU is able to perform about 20 times faster than the CPU. This is because these image processing functions are perfectly suited to the CUDA programming model. Each function operates identi-

cally and independently (i.e., no branching logic) on all the pixels of the image. All memory operations are coalesced. Combining the three functions into one removes a number of global memory operations and allows all the calculations to be done within the registers. All of these factors combine to yield the significant performance improvement observed.

4.7 Modification of Results

After seeing Figure 4.5, one may wonder why each image processing function cannot be executed by the fastest implementation, thereby creating a greater overall speedup. This would be particularly beneficial if the CCL could be executed on the CPU while the other functions are executed on the GPU. There is one potential downfall from using this method. Data must be transferred from the GPU, through the northbridge, and finally is received by the CPU (and vice versa). The time it takes to transfer the image data twice degrades the performance that might be gained by CPU execution. If the time lost from the data transfers is not gained back from fast CPU execution, it would be better to optimize the code to run more efficiently on the GPU.

Slight modifications to the implementations in Table 4.2 can yield significantly different results, as seen in Table 4.3. For example, combining the best CPU functions written in MATLAB and C MEX into one implementation results in an overall speedup of 2.56 times. This means the pure GPU 1.2+ implementation is only 31% faster than the optimized CPU implementation. However, when the slower functions for the GPU are ported back onto the CPU, the GPU has an overall speedup of 5.06 times when compared to the pure MATLAB implementation. Only the first three image processing functions are executed on the GPU and the CCL and Blob Analysis functions are executed on the CPU. This demonstrates the the GPU can significantly improve the performance of applications when it is used properly.

One final observation should be made. In the current MTT implementation, the color information from the camera is never used in the tracking algorithms. Using a

Table 4.3: Summary of the results if different implementations are used for each function. The "Pure CPU" implementation uses only MATLAB and C MEX functions for the image processing. "CPU/GPU" uses the GPU for the first three image processing functions, and then uses the CPU for CCL and Blob Analysis. The overhead times for the GPU are starred because they are estimates.

Function	**Time (ms)**					
	Pure CPU			**CPU/GPU**		
	S1	**S2**	**S3**	**S1**	**S2**	**S3**
Overhead	n/a	n/a	n/a	7.00*	7.00*	7.00*
Color to Grayscale	30.28	27.67	28.46			
Subtract Background	1.77	1.68	1.95	2.45	2.44	2.48
Threshold	3.23	2.80	3.46			
CCL	5.26	4.96	6.15	5.26	4.96	6.15
Blob Analysis	9.47	8.64	11.80	9.47	8.64	11.80
Total Time:	50.01	45.75	51.82	24.18	23.04	27.43
Average Time:		49.19			24.88	
Average FPS:		20.33			40.19	
Average Speedup:		2.56			5.06	

native grayscale video camera would speed up the image processing since it removes the color to grayscale conversion stage. However, this would probably not be desirable in a real application. Feature-aided tracking could use RGB attributes of targets to increase the accuracy of the filtering/prediction and data association.

4.8 Summary

This chapter described the test setup for the MTT software, presented the results of the tests, and then explained any unexpected deviations in the results. The test results only demonstrated a performance improvement for the image processing portion of the MTT software. The other portions of the software were not tested since they do not fit well into the data-parallel model of GPU when there is a small number of targets (less than 100). The GPU is able to run 2.87 times faster than the pure MATLAB implementation. The performance gain on the GPU is greater when it has compute capability 1.2 or higher, since that allows for atomic operations at global memory and also allows for coalesced memory operations for byte-sized variables.

This 287% speedup is significant, considering that it is achieved by using hardware already in most personal computers.

V. Conclusions

This chapter provides a brief summary of the research contributions and outlines a few ideas of future work that can be pursued.

5.1 Research Contribution

There are three main contributions that this research offers. First, it addresses the question of whether a GPU can be used to speed up the image processing portion of MTT. Next, it provides valuable insights into feasible and practical performance gains for MTT and general computing. Finally, a self-contained and reusable MTT implementation is written in MATLAB and portions can be run as C MEX or GPU code.

For AFIT, the self-contained MTT software written in MATLAB will be useful for future research. The software is well documented and highly parameterized, which will make it easier for a researcher to develop additional MTT algorithms in a short time. Instead of developing the entire package, they can focus on the individual portion of MTT they are working on.

Most importantly, this research indicates that the GPU can significantly improve the performance of MTT, especially the image processing. Some functions are improved more than 20 times, while the overall performance improvement is about 3 times. Images were originally able to be processed at 8 FPS, but now (with the GPU) are processed at about 23 FPS with full 24-bit color 1080p HD images. Further optimization of the GPU code may yield even higher speedups.

5.2 Future Work

This research only scratches the surface of augmenting MTT with GPUs. There are many different aspects that can be improved, and other areas of completely new research. Some functions, like CCL and blob analysis, need to be improved to realize further performance gains. Also, different algorithms can be used for image processing to make it more usable in a real-world application. There are also other future research

options that are tangent to this research, namely using a different GPU language, tracking multiple targets in a large area, and tracking targets in a 3D environment.

5.2.1 Function Improvements. One of the least-improved image processing functions in this research is CCL. Time should be taken to develop a more advanced algorithm that efficiently solves the labeling problem on the GPU. Also, algorithms to solve the assignment problem on the GPU should be analyzed for large numbers of targets/observations. This would allow for tracking the targets in real-time.

There are a number of ways to improve the current implementation of CCL. The image should be stored as a 2D array and divided into small sections that can be solved by one block. Before processing the block, the label values for the sub-section (along with the neighboring labels to the outer-perimeter pixels) should be cached into shared memory. A shared flag is used to determine when the labels have all been assigned to the local minimum. Then the perimeter pixels are stored back into memory, all threads are synchronized, and their neighboring labels are re-cached from global memory. This process continues until no more pixels have been relabeled. All memory accesses should be coalesced to ensure maximum efficiency. Just these modifications should improve the performance of the CCL. This and other algorithms need to be researched to see if CCL can be faster on the GPU.

The blob analysis function could also be improved. Right now, each pixel is assigned one thread, and they perform an atomic add to the global memory. Combining the row and column atomic kernels into one would cut in half their memory accesses. Also, since a number kernels are currently used, possibly some of those kernels can be combined in order to decrease the number of global memory accesses. Doing this will also decrease the amount of task switching within each stream-processor core.

In addition, reading the video data could be improved by using prefetching or custom hardware. If the data is prefetched into fast memory, or if custom hardware is able to directly send the data to the CPU or RAM, then the performance of reading in each video frame could be significantly improved.

5.2.2 Image Processing Algorithm Changes. In order to make this code more reliable in real-world applications, the image processing algorithms should be modified. Instead of using background subtraction and thresholding, edge detection should be used to detect targets. This would make the tracking software less susceptible to differences in lighting, and it would not have to continuously calculate/model the background. Also, the video imaging should not be limited to only a stationary camera. Optical flow (and other methods) should be implemented on the GPU to make this viable for other military imaging applications.

5.2.3 Miscellaneous Research Ideas. The code for this project should be ported into the GPU language that dominates the market. Most likely, when OpenCL, described in Section 2.3, is released, it will be a viable candidate since it allows one set of source code to run on any GPU.

Target tracking for a large area (e.g., a city or entire country) could be accomplished by dividing the area into a number of overlapping blocks. Each GPU would process and track an individual grid, and then share the tracking information with neighboring GPUs when the target leaves its tracking area.

There is also potential to solve 3D target tracking problems. A terrain map could be loaded into the GPU texture memory in order to augment the filter predict/update operations.

5.3 Summary

The goal of this thesis research was to determine if a GPU can be used to improve the performance of the image processing for MTT. The results of this research indicate that certain types of image processing map very well to the GPU programming domain. For example, the color to grayscale, background subtraction, and thresholding functions perform about 20 times faster on the GPU than on the CPU. On the other hand, the CCL algorithm is 80% slower on the GPU. Overall,

the image processing is 287% times faster on the GPU than the MATLAB Image Processing Toolbox implementation.

If a faster parallel implementation of CCL and blob analysis can be made, then the performance improvement would increase significantly. The results indicate that more research into MTT on the GPU is worthwhile for potential performance improvements.

Bibliography

1. P. Heckbert, "Color image quantization for frame buffer display," *SIGGRAPH Comput. Graph.*, vol. 16, no. 3, pp. 297–307, 1982.
2. R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam, "Image change detection algorithms: a systematic survey," *IEEE Transactions on Image Processing*, vol. 14, no. 3, pp. 294–307, March 2005.
3. T. F. Fulton, "Change detection for processing of angel fire imagery," Master's thesis, Air Force Institute of Technology, March 2008.
4. M. Piccardi, "Background subtraction techniques: a review," *IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, pp. 3099–3104 vol.4, October 2004.
5. R. C. Gonzalez and R. E. Woods, *Digital Image Processing (2nd Edition)*. Prentice Hall, January 2002.
6. G. Stockman and L. G. Shapiro, *Computer Vision*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.
7. K. Wu, E. Otto, and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis Applications*, 2007.
8. J. Antonakos, "Image processing fundamentals," *Circuit Cellar*, December 2001.
9. S. Blackman and R. Popoli, *Design and Analysis of Modern Tracking Systems*. Artech House, 1999.
10. R. A. Singer, "Estimating optimal tracking filter performance for manned maneuvering targets," *IEEE Transactions on Aerospace and Electronic Systems*, vol. AES-6, no. 4, pp. 473–483, July 1970.
11. T. C. Jansen, "Gpu++ an embedded gpu development system for general-purpose computations," Master's thesis, University of Munich, August 2007.
12. S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 73–82.
13. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.0*, NVIDIA, July 2008.
14. Y. Bar-Shalom, *Tracking and data association*. San Diego, CA, USA: Academic Press Professional, Inc., 1987.

15. P. Konstantinova, A. Udvarev, and T. Semerdjiev, "A study of a target tracking algorithm using global nearest neighbor approach," in *CompSysTech '03: Proceedings of the 4th international conference on Computer systems and technologies*. New York, NY, USA: ACM, 2003, pp. 290–295.

16. P. Konstantinova, M. Nikolov, and T. Semerdjiev, "A study of clustering applied to multiple target tracking algorithm," in *CompSysTech '04: Proceedings of the 5th international conference on Computer systems and technologies*. New York, NY, USA: ACM, 2004, pp. 1–6.

17. Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "Gpucv: an open-source gpu-accelerated framework for image processing and computer vision," in *MM '08: Proceeding of the 16th ACM international conference on Multimedia*. New York, NY, USA: ACM, 2008, pp. 1089–1092.

18. 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.

19. J. Fung and T. Murray, *Photoshop Filters for the GPU*, NVIDIA, April 2008.

20. T. Y. Kong and A. Rosenfeld, Eds., *Topological Algorithms for Digital Image Processing*. New York, NY, USA: Elsevier Science Inc., 1996.

21. M. Manhar and H. K. Ramapriyan, "Connected component labeling of binary images on a mesh connected massively parallel processor," *Comput. Vision Graph. Image Process.*, vol. 45, no. 2, pp. 133–149, 1989.

22. C. Bibby and I. Reid, "Fast feature detection with a graphics processing unit implementation," in *Proceedings of the International Workshop on Mobile Vision*, 2006.

23. MathWorks, "Matlab central - file detail," http://www.mathworks.com/matlab central/fileexchange/6543, January 2009.

24. Wolfram, "Amdahl's law," http://demonstrations.wolfram.com/AmdahlsLaw/, February 2009.

25. B. S. Morse, "Lecture 4: Thresholding," January 2000, Brigham Young University.

Index

The index is conceptual and does not designate every occurrence of a keyword.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2009	Master's Thesis	Sep 2007 — Mar 2009

4. TITLE AND SUBTITLE

Image Processing for Multiple-Target Tracking on a Graphics Processing Unit

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Michael Allen Tanner, 2d Lt, USAF

5d. PROJECT NUMBER

09-248

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/09-11

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Labs (AFMC)
Dr. Devert Wicker
2241 Avionics Circle
WPAFB, OH 45433-7765
(937–674–9871; devert.wicker@wpafb.af.mil)

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYAT

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Multiple-target tracking (MTT) systems have been implemented on many different platforms, however these solutions are often expensive and have long development times. Such MTT implementations require custom hardware yet offer very little flexibility with ever changing data sets and target tracking requirements. This research explores how to supplement and enhance MTT performance with an existing graphics processing unit (GPU) on a general computing platform. Typical computers are already equipped with powerful GPUs to support various games and multimedia applications. However, such GPUs are not currently being used in desktop MTT applications.
Bottleneck MTT image processing functions (frame differencing) were converted to execute on the GPU. On average, the GPU code executed 287% faster than the MATLAB implementation. Some individual functions actually executed 20 times faster than the baseline. These results indicate that the GPU is a viable source to significantly increase the performance of MTT with a low-cost hardware solution.

15. SUBJECT TERMS

Target Tracking, Kalman Filter, Graphics Processing Unit, Blob Analysis, Connected Component Labeling

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Yong Kim
U	U	U	UU	79	19b. TELEPHONE NUMBER *(include area code)* 937–255–3636, ext 4620; ykim@afit.edu