Speeding Up Particle Filter Algorithm for Tracking Multiple Targets Using CUDA Programming

Jinhua Zhang

Marquette University

Follow this and additional works at: https://epublications.marquette.edu/theses_open

Recommended Citation
https://epublications.marquette.edu/theses_open/598
ABSTRACT

SPEEDING UP PARTICLE FILTER ALGORITHM FOR TRACKING MULTIPLE TARGETS USING CUDA PROGRAMMING

Jinhua Zhang, B.S.
Marquette University

This thesis proposes to work on a parallelization method to speed up the computational runtime of the particle filter algorithm for multiple targets tracking. CUDA programming is utilized to execute the original implementation of the particle filter algorithm on GPU. The thesis provides a detailed discussion of the background information on the relevant topics. And then a presentation of the code architecture changes is followed. The detailed CUDA-based implementation is illustrated and discussed, which is followed by a discussion and comparison of the results obtained from a series of tests.

In this thesis, the introduction and description of the basic particle filter are presented first. Detailed illustrations of each step in the original implementation of the particle filter algorithm, which is executed sequentially on CPU, are provided. Then, background information of parallel programming technologies is provided, such as GPGPU and CUDA programming. The new design of the CUDA based implementation of the particle filter algorithm is proposed to speed up the execution of the original implementation, which is executed on CPU. Moreover, a detailed explanation of the CUDA-based implementation is given.

Finally, the thesis will demonstrate the test results for both CPU and CUDA implementation as a comparison. The experiments indicate that the CUDA implementation can obtain a maximum of 7.5x speedup over the original implementation. After implementing more results and comparison, it was concluded that the CUDA implementation was significantly faster than the CPU version. Furthermore, the CUDA version still has much space for future optimizations to increase its performance.
ACKNOWLEDGEMENTS

Firstly, I would like to thank my parents who have been supporting me for the past 25 years and giving encouragement. Secondly, I would like to thank my advisor, Dr. Cristinel Ababei, who leads me to the path of being a Master student. He is always patient and positive to answer my questions. It was a great honor to work with him in the past one and half years. I learned a lot from him not only the way to think but also the way to work. He is a competent mentor and professor. I will never regret deciding on researching with Dr. Cristinel Ababei. I also would like to thank my other committee members, Dr. Richard J. Povinelli and Dr. Henry Medeiros, for being my committee members and taking the time to review the thesis and provide the meritorious feedback. I would also like to thank all of my mates in Mess Lab at Marquette University. They generated endless encouragement and support to me. When I have difficulties in my life and study, they are always willing to give me their hands. Finally, I would like to thank my girlfriend Yuxin Ji who has been staying with me for the hardest time and always showing me her endless love.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ......................................................... i

TABLE OF CONTENTS ...................................................... ii

LIST OF TABLES ............................................................ iv

LIST OF FIGURES ............................................................ v

CHAPTER 1 Problem Statement, Objective and Contributions . 1
  1.1 Problem Statement .................................................. 1
  1.2 Objectives ........................................................... 2
  1.3 Contributions ....................................................... 3
  1.4 Thesis Organization ................................................ 3

CHAPTER 2 Background on the Particle Filter Algorithm ....... 4
  2.1 Background Information ........................................... 4
  2.2 Description of Particle Filter Algorithm ....................... 5
    2.2.1 Bayesian Estimation ......................................... 5
    2.2.2 Sequential Importance Sampling ......................... 10
    2.2.3 Degeneracy Problem ........................................ 13
    2.2.4 Resampling .................................................. 15
  2.3 The Implementation of the Particle Filter Algorithm on CPU 16
  2.4 Computational Complexity of the Sequential Implementation 20
  2.5 Summary ........................................................... 22

CHAPTER 3 Related Work .................................................. 23
  3.1 Summary ........................................................... 32

CHAPTER 4 Parallel Processing Technologies ....................... 33
  4.1 Basics of Parallel Computing ..................................... 33
    4.1.1 Serial Computing ........................................... 33
    4.1.2 Parallel Computing ......................................... 33
    4.1.3 GPGPU ....................................................... 35
  4.2 CUDA Programming ................................................ 36
    4.2.1 Host Code .................................................. 38
4.2.2 Device Code ................................................. 39
4.3 Summary ....................................................... 40

CHAPTER 5 CUDA-Based Implementation of the Particle Filter 41
5.1 The Flowchart of the CUDA-Based Implementation of the Particle
Filter Algorithm ..................................................... 41
5.2 Detailed Description of The CUDA-based Implementation ............. 42
5.2.1 Preparatory Work .............................................. 43
5.2.2 Kernel Implementation ....................................... 48
5.3 Summary ......................................................... 55

CHAPTER 6 Discussion of Results ................................. 59
6.1 Results from Testing Different Video Frame Sizes ................. 60
6.2 Results for Different Numbers of Particles .......................... 64
6.3 Results for the Real-time Video Streams in Dynamic Surveillance
Scenario ................................................................. 69
6.4 Results for the Bolt Dataset ...................................... 70
6.5 Further Observations ............................................ 74
6.6 Summary ......................................................... 75

CHAPTER 7 Conclusion and Future Work .......................... 76
7.1 Conclusions ..................................................... 76
7.2 Future Work ................................................... 77

REFERENCES ......................................................... 79
LIST OF TABLES

2.1 The concrete data of the variation of the Execution Time of the particle filter algorithm with respect to the number of particles when executed sequentially on a CPU. 21

3.1 Comparison of the proposed work to previous studies [Part1] 30

3.2 Comparison of the proposed work to previous studies [Part2] 31

6.1 Execution times of main steps of the algorithm. 73
LIST OF FIGURES

2.1 An example of using the particle filter algorithm to track persons. . . . 5
2.2 The flowchart of Bayesian estimation. . . . . . . . . . . . . . . . . . . . 7
2.3 An illustration the degeneracy problem. . . . . . . . . . . . . . . . . . 14
2.4 Flowchart of the sequential particle filter algorithm. . . . . . . . . . . 17
2.5 The initialization of each tracking object under Commander class. . . 18
2.6 The initialization of particles for each object by the constructor of Police class. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.7 The initialization of parameter information for each particle by the constructor of Dog class. . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.8 Portion of the gprof output profile that shows the function calls that take most of the execution time. . . . . . . . . . . . . . . . . . . 20
2.9 Variation of the Execution Time of the particle filter algorithm with respect to the number of particles when executed sequentially on a CPU. 21
4.1 Illustration of serial computing. . . . . . . . . . . . . . . . . . . . . . 34
4.2 Illustration of parallel computing. . . . . . . . . . . . . . . . . . . . . 34
4.3 CUDA example of vector adding [1]. . . . . . . . . . . . . . . . . . . . 37
4.4 Device code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.1 Illustration of the CUDA-based implementation of the particle filter algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2 Function that initializes DogArr and prepares to store particles information obtained from Dog Vector. . . . . . . . . . . . . . . . . . 44
5.3 Function to allocate memory for the particles on GPU. .................. 45

5.4 Function to fill in the arrays with the predicted particles. ............... 46

5.5 Function that copies predicted particles from host RAM to device global
memory. ........................................................................... 47

5.6 Function to copy images to the array. ....................................... 48

5.7 Function to transfer the images to device. ................................. 48

5.8 Function that calculates Hue histogram for each predicted particle and
assigns the weight based on its Bhattacharyya distance with respect to
the original particle. ............................................................... 49

5.9 CUDA kernel implementation. ............................................... 50

5.10 Likelihood function on device. .............................................. 51

5.11 Function that calculates of Hue histogram on CPU. .................... 52

5.12 Function to calculate of the Hue histogram on device. ............... 56

5.13 Functions to convert RGB to HSV on device. .......................... 57

5.14 Function that calculates of Hue Bhattacharyya distance between the
predicted particles. ............................................................... 58

6.1 Variation of the execution time of the particle filter algorithm using
CUDA and CPU implementations with respect to different video resolu-
tions, for 1024 particles. ...................................................... 61

6.2 Variation of the execution time of the particle filter algorithm using
CUDA and CPU implementations with respect to different video resolu-
tions, for 3072 particles. ...................................................... 62
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 6144 particles.</td>
<td>62</td>
</tr>
<tr>
<td>6.4</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 9216 particles.</td>
<td>63</td>
</tr>
<tr>
<td>6.5</td>
<td>The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different video resolutions.</td>
<td>63</td>
</tr>
<tr>
<td>6.6</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 640 x 360 video resolution.</td>
<td>65</td>
</tr>
<tr>
<td>6.7</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 856 x 480 video resolution.</td>
<td>65</td>
</tr>
<tr>
<td>6.8</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 1280 x 720 video resolution.</td>
<td>66</td>
</tr>
<tr>
<td>6.9</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 1920 x 1080 video resolution.</td>
<td>66</td>
</tr>
<tr>
<td>6.10</td>
<td>Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 3840 x 2160 video resolution.</td>
<td>67</td>
</tr>
<tr>
<td>6.11</td>
<td>The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different numbers of the particles.</td>
<td>68</td>
</tr>
</tbody>
</table>
6.12 Selected frames that show tracking of a prerecorded video, using the CUDA implementation .................................................. 68

6.13 Four frames selected from the CUDA-based implementation with face detection. ................................................................. 69

6.14 Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different particle numbers, on the OTB100 benchmark. ........................................ 71

6.15 The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different numbers of the particles. ......................................................... 71

6.16 Tracking results on the Bolt dataset from OTB100 benchmark. .... 74
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU:</td>
<td>Central Processing Units.</td>
</tr>
<tr>
<td>CUDA:</td>
<td>Compute Unified Device Architecture.</td>
</tr>
<tr>
<td>GPU:</td>
<td>Graphics Processing Units.</td>
</tr>
<tr>
<td>GPGPU:</td>
<td>General Purpose Computing on Graphics Processing Units.</td>
</tr>
<tr>
<td>FPGA:</td>
<td>Field Programmable Gate Array.</td>
</tr>
<tr>
<td>DSP:</td>
<td>Digital Signal Processor.</td>
</tr>
<tr>
<td>SIS:</td>
<td>Sequential Importance Sampling.</td>
</tr>
<tr>
<td>MC:</td>
<td>Monte Carlo.</td>
</tr>
<tr>
<td>HLS:</td>
<td>High-Level Synthesis.</td>
</tr>
<tr>
<td>CORDIC:</td>
<td>Coordinate Rotation Digital Computer.</td>
</tr>
<tr>
<td>RNG:</td>
<td>Random Number Generator.</td>
</tr>
<tr>
<td>SIMD:</td>
<td>Single Instruction Multiple Data.</td>
</tr>
</tbody>
</table>
CHAPTER 1

Problem Statement, Objective and Contributions

1.1 Problem Statement

The problem addressed in this thesis is the long execution time of the particle filter algorithm executed on sequential Central Processing Units (CPUs). The problem is solved by the parallelization of the particle filter algorithm using Compute Unified Device Architecture (CUDA) programming, targeting CUDA-capable Graphics Processing Units (GPU) architectures. In the past 30 years, the majority of the algorithms were written only to exploit a single core [2]. It was a reasonable approach since most of the consumer CPUs in personal computers only had a single core. The processor ran with a significantly high frequency, which provided reasonable performance for many computing problems. Today, even low-end, low-power central processors have two or more cores per die. Therefore, it is natural to attempt to develop algorithms to take benefit of such parallel processors. Multithreading is regarded as the ability of CPUs to execute multiple threads concurrently. Multithreading decreases execution time by splitting different tasks up, which are then executed by different threads on the CPU. In 2006, NVIDIA introduced CUDA, a general-purpose parallel computing platform and a programming model for GPUs, which can help solve many complex computational problems more efficiently compared to CPUs [2].

This thesis proposes to speed up the particle filter algorithm for tracking multiple targets using CUDA programming. The thesis first discusses the background information on the advantage of the particle filter algorithm in
object tracking compared to other algorithms and the advantage of parallel processing, specifically CUDA programming, for speeding up parallel computation tasks. The remainder of the thesis focuses on how to meet the proposed performance goals. An illustration of the original implementation of the particle filter algorithm that used OpenCV libraries and executed on general-purpose CPUs is presented. Next, a parallel implementation is proposed to improve the performance of the existing algorithm. To that end, we propose to use CUDA programming targeting NVIDIA’s General Purpose Computing on Graphics Processing Units (GPGPUs). Then, we also demonstrate it on real-time videos. Finally, the thesis analyzes and compares the performance results of the traditional algorithm of the implementation on CPU and the CUDA based implementation of the particle filter algorithm.

1.2 Objectives

A primary goal of the thesis is to present an approach using CUDA programming to accelerate the particle filter algorithm for tracking objects in real-time videos. The CUDA based implementation will be developed starting from an open-source project [3] of the particle filter algorithm that used OpenCV libraries. The original program was only single-threaded and executed serially on CPUs. Several types of experiments are conducted to investigate and to validate the CUDA based implementation of the particle filter algorithm.
1.3 Contributions

This thesis provides a method to parallelize using CUDA programming the CPU sequential implementation of the particle filter algorithm. The key contributions of this thesis are as follows:

- Describe the implementation with CUDA programming of the particle filter algorithm. The focus is on the modified portions of the original code.
- Conduct comprehensive experiments to investigate and validate the scalability of the CUDA based implementation.
- Demonstrate the CUDA implementation on real-time video streams from a camera connected to the computer.
- Provide the CUDA based implementation of the particle filter algorithm as an open-source code repository.
- Discuss potential future improvements to the CUDA based implementation.

1.4 Thesis Organization

Chapter 2 presents an overview of the particle filter algorithm and describes the related work. Chapter 3 describes the related work. Chapter 4 introduces the background information about parallel computing and CUDA programming. Chapter 5 describes the CUDA based parallelization of the particle filter algorithm. Chapter 6 provides a comparison of the results determined with implementation on different processing platforms, including on real-time video streams. Chapter 7 concludes the thesis and discusses ideas for future work.
CHAPTER 2

Background on the Particle Filter Algorithm

In this chapter, we first provide an overview of the particle filter algorithm. Then, the current limitations of the particle filter algorithm and challenges for tracking objects are discussed. The primary motivation of this thesis is the long computational runtime of the particle filter algorithm. In the third section, we illustrate the original sequential implementation of the particle filter algorithm. Finally, we profile the algorithm in order to identify the portion that dominates the computational time.

2.1 Background Information

Object tracking is an essential task in the field of modern video surveillance. The traditional implementation of tracking algorithms for video streams has been done typically on prerecorded videos. On the contrary, surveillance systems require the camera to implement tracking algorithms, such as the particle filter algorithm, on live video for real-time analysis, for example, for person detection and tracking [4] shown in Fig. 2.1. However, many tracking algorithms require high computational resources. Therefore, the processing of these algorithms is often implemented via parallelization approaches. Field Programmable Gate Array (FPGAs), GPGPUs, and Digital Signal Processors (DSPs) were used as parallelization solutions [5]. In this thesis, an approach for speeding up the particle filter algorithm using CUDA programming is proposed.
2.2 Description of Particle Filter Algorithm

The particle filter algorithm is a signal processing method, which has the advantage that it can handle non-linear and non-Gaussian systems [6]. The particle filter algorithm combines Bayesian estimation and sequential Monte Carlo Sampling with better performance for non-linear and non-Gaussian systems. That includes surveillance, object tracking, computer, and robot vision, widely used in real-world applications [5]. The particle filter algorithm has been shown to perform better compared to Kalman filters due to its robustness and effectiveness [7, 8].

2.2.1 Bayesian Estimation

For estimation problems, we use the following state model of state space representation to estimate the state of a dynamic system changing over time [5]:

Figure 2.1: An example of using the particle filter algorithm to track persons.
$x_k = f_{k-1}(x_{k-1}, w_{k-1}), \quad (2.1)$

where $f_{k-1}$ is a possibly nonlinear function of the state $x_k$, and $w_{k-1}$ is the process noise. Discrete-time steps are represented by $k$. The measurement model describes the measurement of a dynamic system, as follows:

$y_k = h_k(x_k, v_k), \quad (2.2)$

where $h_k$ is a possibly nonlinear function. $y_k$ is the measurement vector and $v_k$ is the measurement noise. From the standpoint of Bayesian estimation, state estimation problems such as object tracking and signal filtering are solved by calculating the posterior probability density function (pdf) $p(x_k|y_{1:k})$ recursively via the measurement vector $y_k$ and the prior pdf $p(x_k|y_{1:k-1})$. This estimation process is specifically done in two stages: Prediction and Update, which are described next. Fig. 2.2 describes the simple process of Bayesian estimation including Prediction and Update stages from time $k-1$ to time $k$.

1) Prediction. Suppose the pdf $p(x_{k-1}|y_{1:k-1})$ at time $k-1$ is known, then, the prior pdf $p(x_k|y_{1:k-1})$ at time $k$ is obtained by using the state model equation 2.1 and the Chapman-Kolmogorov equation [9], as shown in equation 2.3
Figure 2.2: The flowchart of Bayesian estimation.

\[
p(x_k|y_{1:k-1}) = \int p(x_k|x_{k-1}, y_{1:k-1}) p(x_{k-1}|y_{1:k-1}) \, dx_{k-1}
\]

\[
= \int p(x_k|x_{k-1}) p(x_{k-1}|y_{1:k-1}) \, dx_{k-1}.
\]

(2.3)
Generally, the state transition of the system is assumed to follow the first-order Markov model. That means that the current state $x_k$ is only dependent on the state of previous time step $x_{k-1}$. The fact that $p(x_k|x_{k-1}, y_{1:k-1}) = p(x_k|x_{k-1})$ has been derived from this assumption. The pdf $p(x_k|x_{k-1})$ is the transition model that generally is assumed. In this thesis, we use a second-order, auto-regressive dynamical model which is give in equation 2.4:

$$x_k - \bar{x} = A_2(x_{k-2} - \bar{x}) + A_1(x_{k-1} - \bar{x}) + B_0w_k.$$  \hspace{1cm} (2.4)

This equation predicts the next state based on the previous two plus some noise $w_k$ that is generated from a Gaussian distribution.

2) Update. At time $k$, the measurement $y_k$ is known, which can be used to update the prior pdf by the Bayesian equation:

$$p(x_k|y_{1:k}) = \frac{p(y_k|x_k)p(x_k|y_{1:k-1})}{p(y_k|y_{1:k-1})}. \hspace{1cm} (2.5)$$

As shown in equation 2.5, the posterior pdf $p(x_k|y_{1:k})$ is described by three terms [9]: the likelihood function $p(y_k|x_k)$, the prior pdf $p(x_k|y_{1:k-1})$ given by the Prediction stage, and $p(y_k|y_{1:k-1})$ given in equation 2.6:

$$p(y_k|y_{1:k-1}) = \int p(y_k|x_k, y_{1:k-1})p(x_k|y_{1:k-1}) \, dx_k$$

$$= \int p(y_k|x_k)p(x_k|y_{1:k-1}) \, dx_k. \hspace{1cm} (2.6)$$
The likelihood function \( p(y_k|x_k) \) is defined by the measurement model given by equation 2.2 and known as the measurement noise \( v_k \), which is the observation model that specifics the likelihood of an object being a specific state (i.e. at a specific location). In this thesis, we use a simple HSV histogram model. Equation 2.6 is given by applying Chapman-Kolmogorov equation which is shown in equation 2.7:

\[
p(y|x) = \frac{p(x,y)}{p(x)} = \int \frac{p(x,y,z)dz}{p(x)} = \int p(y|x,z)p(z|x)p(x)dz = \int p(y|x,z)p(z|x)dz.
\] (2.7)

In the Update phase, the posterior pdf \( p(x_k|y_{1:k}) \) at time \( k \) is obtained by modifying the prior pdf \( p(x_k|y_{1:k-1}) \) with measurement vector \( y_k \). The two recurrence relations given by the equations 2.3 and 2.5 represent the basis of the Bayesian estimation [9]. From the above equations, we can have the conceptual solution of the posterior pdf \( p(x_k|y_{1:k}) \). However, the recursive propagation of posterior pdf is only a conceptual solution. The solution cannot be determined analytically for a nonlinear and non-Gaussian system. In order to solve this problem, a sequential importance sampling (SIS) using Monte Carlo (MC) method is introduced here to obtain an equivalent representation of posterior pdf for the optimal Bayesian estimate.
2.2.2 Sequential Importance Sampling

Sequential importance sampling is regarded as a MC method to estimate the required posterior pdf $p(x_k|y_{1:k})$ by a set of random samples with associated weights $[6,10]$. MC simulations compute the estimates based on the samples and weights. The estimate consists of an equivalent representation to the functional description of the posterior pdf.

The following explanation of sequential importance sampling is adapted from the study in [9]. Suppose we have a set of random particles $\{x^i_{0:k}, w^i_k\}_{i=1}^{N_s}$ sampled from the posterior pdf $p(x_k|y_{1:k})$, where $\{x^i_{0:k}, i = 0, ..., N_s\}$ is a set of support particles with associated weights $\{w^i_k, i = 0, ..., N_s\}$ and the set of all state steps to time $k$ is $x_{0:k} = \{x_j, j = 0, ..., k\}$ [9]. The weights are normalized to values in the range $(0,1)$. The posterior pdf can be estimated as follows:

$$p(x_{0:k}|y_{1:k}) \approx \sum_{i=1}^{N_s} w^i_k \delta(x_{0:k} - x^i_{0:k}).$$

(2.8)

The weights $w^i_k$ are decided by importance sampling [11,12]. Specifically, suppose $p(x) \propto \pi(x)$ and $\pi(x)$ is a probability density from which $p(x)$ is difficult to sample, because it is posterior pdf. However, $\pi(x)$ as a probability can be evaluated and $p(x)$ is up to proportionality of $\pi(x)$. In order to solve this problem, a proposal $q(x)$ called an importance density is proposed. $x^i$ can be easily sampled from it. Therefore, an approximation to pdf $p(x)$ is described by:
\[ p(x) \approx \sum_{i=1}^{N_s} w^i \delta(x - x^i). \quad (2.9) \]

\[ w^i \propto \frac{\pi(x^i)}{q(x^i)}. \quad (2.10) \]

Equation 2.10 presents the normalized weight of the \( i \)th particle. Hence, the weights in equation 2.8 can be defined to be:

\[ w_k^i \propto \frac{p(x_{0:k}^i | y_{1:k})}{q(x_{0:k}^i | y_{1:k})}. \quad (2.11) \]

Now, the problem that it is difficult to sample from posterior density function has been solved by sampling particles from \textit{importance density} \( q(x) \).

Back to the recursive derivation of weights, we suppose that the \( q(x_{0:k}^i | y_{1:k}) \) estimates the posterior density of all the states of the time up to \( k \). The importance density can be factorized as:

\[ q(x_{0:k} | y_{1:k}) = q(x_k | x_{0:k-1}, y_{1:k}) q(x_{0:k-1} | y_{1:k-1}). \quad (2.12) \]

The recursive propagation of weights can be defined as follows:
\begin{align*}
p(x_{0:k}|y_{1:k}) &= \frac{p(y_k|x_{0:k}, y_{1:k-1})p(x_{0:k}|y_{1:k-1})}{p(y_k|y_{1:k-1})} \\
&= \frac{p(y_k|x_{0:k}, y_{1:k-1})p(x_k|x_{0:k-1}, y_{1:k-1})}{p(y_k|y_{1:k-1})} \\
&\quad \times p(x_{0:k-1}|y_{1:k-1}) \\
&= \frac{p(y_k|x_k)p(x_k|x_{k-1})}{p(y_k|y_{1:k-1})} p(x_{0:k-1}|y_{1:k-1}) \\
&\propto p(y_k|x_k)p(x_k|x_{k-1})p(x_{0:k-1}|y_{1:k-1}).
\end{align*}

The weight equation can be defined as in equation 2.14 by substituting equations 2.12 and 2.13 into equation 2.11.

\begin{align*}
w^i_k &\propto \frac{p(y_k|x^i_k)p(x^i_k|x^i_{k-1})p(x^i_{0:k-1}|y_{1:k-1})}{q(x^i_k|x^i_{0:k-1}, y_{1:k})q(x^i_{0:k-1}|y_{1:k-1})} \\
&= w^i_{k-1} \frac{p(y_k|x^i_k)p(x^i_k|x^i_{k-1})}{q(x^i_k|x^i_{0:k-1}, y_{1:k})}.
\end{align*}

In addition, if \(q(x^i_k|x^i_{0:k-1}, y_{1:k}) = q(x^i_k|x^i_{k-1}, y_k)\), the importance density is only decided by \(x_{k-1}\) and \(y_k\). Therefore, the weight given by equation 2.14 can be defined as:

\begin{align*}
w^i_k &\propto w^i_{k-1} \frac{p(z_k|x^i_k)p(x^i_k|x^i_{k-1})}{q(x^i_k|x^i_{k-1}, y_k)},
\end{align*}

and the posterior density \(p(x_k|y_{1:k})\) can be approximated as:

\begin{align*}
p(x_k|y_{1:k}) &\approx \sum_{i=1}^{N_S} w^i_k \delta(x_k - x^i_k).
\end{align*}
The weights $w_i^k$ are defined as in equation 2.15. The estimation is close to true posterior density $p(x_k|y_{1:k})$. A pseudocode description of the sequential importance sampling particle filter algorithm is given in Algorithm 1 listed below. $N_s$ particles are sampled from $q(x_k|x_{k-1}^i, y_k)$ and assigned weights recursively according to equation 2.15.

**Algorithm 1 Sequential Importance Sampling Particle Filter**

1: $\{\{x_{k}^{i}, w_{k}^{i}\}_{i=1}^{N_s}\} = \{\{x_{k-1}^{i}, w_{k-1}^{i}\}_{i=1}^{N_s}, y_k\}$

2: for $i = 1$ to $N_s$ do
3:   Draw $x_k^i$ from $q(x_k|x_{k-1}^i, y_k)$
4:   Assign the particle a weight, $w_k^i$ according to equation 2.15
5: end for

### 2.2.3 Degeneracy Problem

A common problem of the SIS particle filter is the degeneracy phenomenon. That is, after a few iterations, all of the particles will have negligible weights, and only one particle remains with a large weight compared to others [12]. Besides, the variance of the importance weights increases over time. When the number of particles whose contribution becomes insignificant, most of the computation will be wasted on that. That significantly decreases the performance of the approximation, as shown in Fig. 2.3. In this figure, the sizes of the dots represent the weight of the particle, and after a few iterations, most of the particles have negligible weights, only one particle remains with a large weights. An appropriate measure of degeneracy is the effective sample size $N_{eff}$ described in [11,13].
Figure 2.3: An illustration the degeneracy problem.

\[ N_{\text{eff}} = \frac{N_s}{1 + \text{Var}(w^i_k)}, \quad (2.17) \]

where \( w^i_k \) is the true weight, which cannot be evaluated exactly. So, an estimate \( \hat{N}_{\text{eff}} \) of \( N_{\text{eff}} \) is obtained by:

\[ \hat{N}_{\text{eff}} = \frac{1}{\sum_{i=1}^{N_s} (w^i_k)^2}, \quad (2.18) \]

where \( w^i_k \) is the normalized weight obtained using equation 2.14. Equation 2.18 indicates that when the number of effective particles is small, the variance of weights will be large. As a result, the degeneracy tends to be severe. In order to solve the degeneracy problem, the resampling method has been introduced, which is described in the next section.
2.2.4 Resampling

The main idea of resampling is to eliminate the particles that have small weights and compute the particles with large weights. A new set \( \{x^{i*}_k\}^N_{i=1} \) is generated by resampling \( N_s \) times from the approximation \( p(x_k|y_{1:k}) \) defined by:

\[
p(x_k|y_{1:k}) \approx \sum_{i=1}^{N_s} w^i_k \delta(x_k - x^i_k).
\]  

(2.19)

More specifically, the particle with the largest weight creates the most copies. The particle with the second-largest weight creates the second-largest number copies. The rest can be processed in a similar manner. After resampling, the basic particle filter algorithm is complete. The pseudocode of the particle filter algorithm is described in Algorithm 2.

**Algorithm 2 The Particle Filter**

1: for each \( i = 0 \) to \( N \) do
2:  Particle \( x^{(i)}_0 \) from initial prior \( p(x_0) \)
3: end for
4: for each time step \( k \) do
5:  for each \( i = 0 \) to \( N \) do
6:   \( x^{(i)}_k \leftarrow p(x_k|x_{k-1}^{(i)}) \)
7:   Weight \( w^i_k \leftarrow p(y_k|x^i_k) \)
8:   \( w^i_k \leftarrow w^i_k/\sum_n w^n_k \)
9:  [\{x^i_k, w^i_k, -\}^i_{i=1}] = Resample [\{x^i_k, w^i_k\}^i_{i=1}] - 
10: end for
11: Estimate the state \( x_k = \sum_{i=1}^{N} x^i_k w^i_k \)
12: end for

In Algorithm 2, at time \( k = 0 \), the particles are initialized according to the prior pdf \( p(x^{(i)}_0) \). And at each time step from 1 to \( N \), the particles are sampled from the importance density \( p(x_k|x_{k-1}^{(i)}) \) and assigned weights. Then the weights
of the particles are normalized. Resampling as described in equation 2.19 is presented to generate the new set of the particles.

2.3 The Implementation of the Particle Filter Algorithm on CPU

In this section, we discuss the initial implementation of the particle filter algorithm that is used in the OpenCV libraries and was intended to be executed on general-purpose CPUs. The objects in prerecorded video files can be tracked accurately with relatively low latency for cases when the number of particles is small. However, when the number of particles is increased, the execution time increases significantly. Next, we describe the main steps of the implementation executed on CPUs with the help of Fig. 2.4. In addition, some portions of the code are discussed for a better understanding of the original implementation on CPU.

1) Initialization. Once the first frame of the video is read, the objects to be tracked are selected by users manually, and then the particles are initialized in the center of the rectangle region, which is created by users. In the case of dynamic or real-time detection, manual selection is replaced with an automatic method to detect the object of interest, such as a face for face detection applications. Each particle is generated with its own weight and its location information. In our implementation, the particles are initialized in the center of the region of interest (ROI) with zero weight [14]. Next, the initial likelihood for each particle is calculated by computing the Hue histogram of the ROI. Because all the particles have the same location information, the initial likelihoods of all
particles are the same. In the sequential implementation of the particle filter algorithm on CPU, the model is processed by the *commander* function, which controls all the processes including *Initialization*, *Transit* and *Resampling*. The particle will be assigned to the object to track. The Figs. 2.5, 2.6, and 2.7 describe the original code of the *Initialization* session that we discussed above. Once all the particles are initialized, the algorithm moves to the next step, the *Transition*.

Figure 2.4: Flowchart of the sequential particle filter algorithm.
1. Assign one police to handle one tracking object and store all the polices in the vector _polices

```cpp
to handle one tracking object and store all
the polices in the vector _polices
```

```cpp
void Commander::initialize(const cv::Mat &frame, const std::vector<cv::Rect> &pv)
{
  std::vector<cv::Rect>::const_iterator it;
  for (it = pv.begin(); it != pv.end(); it++)
  {
    Police *pp = new Police(frame, *it, NUMDOGS_PER_POLICE);
    _polices.push_back(pp);
  }
}
```

Figure 2.5: The initialization of each tracking object under Commander class.

2. Assign each police (object) with many dogs (particles), and initialize the parameters information for each particles

```cpp
Police::Police(const cv::Mat &frame, const cv::Rect &rect, const int num_dogs)
{
  ColorHistogram ch;
  cv::Mat imgROI = frame(rect);
  cv::Mat *hist = ch.getHueHistogram(imgROI, 40);
  for (int i = 0; i < num_dogs; i++)
  {
    Dog *pd = new Dog(frame.cols, frame.rows, rect, hist);
    _dogs.push_back(pd);
    _num_dogs = num_dogs;
  }
}
```

Figure 2.6: The initialization of particles for each object by the constructor of Police class.

2) Transition. During the Transition, all the initial particles are updated to be the predicted particles. The predicted particles are assigned randomly to the locations generated from a Gaussian distribution as to fill the image. The predicted particles assigned outside the image are restricted inside the frame through scaling. During the next step, the calculation of the likelihood function for each predicted particle presenting at the object region is calculated by computing its new Hue histogram of the rectangle region in which the predicted
// Create the relevant parameters for each particles (dog)
Dog::Dog(const int fw, const int fh, const cv::Rect &rect, cv::Mat *hist)
{
    this->fw = fw;
    this->fh = fh;
    this->x0 = this->xp = this->x = rect.x + rect.width / 2;
    this->y0 = this->yp = this->y = rect.y + rect.height / 2;
    this->sp = 1.0;
    this->s = 1.0;
    this->width = rect.width;
    this->height = rect.height;
    this->hist = hist;
    this->weight = 0;
}

Figure 2.7: The initialization of parameter information for each particle by the constructor of Dog class.

The location is assumed to be the center of the region. The Bhattacharyya Distance in Hue histograms of rectangle regions between initial particles and predicted particles decide the weight of the predicted particles. More specifically, if the distance is small, then the predicted particles are assumed to be very close to the tracked object and therefore, they are given large weights. In contrast, the particles that are far from the tracked object are given small weights.

3) Normalization. The weights of the predicted particles are normalized within the (0,1) range and then sorted from the largest to the smallest.

4) Resampling. Based on the weights computed before, the predicted particles with larger weights are copied. Specifically, the number of the copies of predicted particles is decided by their percentages of weights in the total weight. The particle with the largest weight is copied the largest number of times. The particle with the second-largest has the second-largest copies. The rest can be
2.4 Computational Complexity of the Sequential Implementation

Before using CUDA programming to parallelize the particle filter algorithm, we identified the portions of the algorithm that dominate the computational runtime. We profiled the original implementation on a Linux machine with a profiler tool called gprof [15] which allows programmers to learn where the program spent its time and which functions called which other functions while it was executing. By analyzing gprof’s output file shown in Fig. 2.8, we found that approximately 80% of the execution time is taken by the calculation of the likelihood function, which computes the histograms and weights of the particles. The algorithm computes the histogram for a rectangle region of pixels formed by a particle, and this must be done separately for all particles. Thus, when the video resolution and the number of particles increase, the computational runtime is dominated by this portion of the overall particle filter algorithm. That is why our implementation will specifically focus on that portion of the algorithm.

Fig. 2.9 and Table 2.1 show the variation of the execution time of the
Figure 2.9: Variation of the Execution Time of the particle filter algorithm with respect to the number of particles when executed sequentially on a CPU.

<table>
<thead>
<tr>
<th>Video Resolutions</th>
<th>1024 Particles</th>
<th>3072 Particles</th>
<th>6144 Particles</th>
<th>9216 Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>360P</td>
<td>1.503</td>
<td>1.7269</td>
<td>2.0641</td>
<td>2.3781</td>
</tr>
<tr>
<td>480P</td>
<td>3.7068</td>
<td>7.8876</td>
<td>14.2961</td>
<td>20.7397</td>
</tr>
<tr>
<td>720P</td>
<td>6.1271</td>
<td>14.577</td>
<td>27.4292</td>
<td>40.0471</td>
</tr>
<tr>
<td>1080P</td>
<td>11.0366</td>
<td>27.28</td>
<td>50.9217</td>
<td>75.434</td>
</tr>
<tr>
<td>2K</td>
<td>21.8365</td>
<td>57.2425</td>
<td>109.9543</td>
<td>163.2318</td>
</tr>
</tbody>
</table>

Table 2.1: The concrete data of the variation of the Execution Time of the particle filter algorithm with respect to the number of particles when executed sequentially on a CPU.

The initial particle filter algorithm on a traditional CPU. In the figure 2.9, we can see that the execution time increases dramatically with the increase in the number of particles. The total execution time of the test video without tracking is 3 seconds. However, in Table. 2.1 when the video resolution is increased to 1280 x 1080 pixels with 9216 particles, the execution time increases to 75 seconds. To address the problem of this poor scalability, in this thesis, we propose to use CUDA programming to improve the computational speeds for larger video
resolutions and numbers of particles.

2.5 Summary

In this chapter, we discussed the theory behind the particle filter algorithm, which combines Bayesian estimation and sequential importance sampling. Also, the resampling method is discussed because it used as a technique to address the degeneracy problem. The original implementation [3] (that is used a reference or base case in this thesis) of the particle filter algorithm on CPUs was described in detail then. Before implementing the particle filter algorithm on GPUs using CUDA programming, we profiled the algorithm to find out the portion that dominated the execution time and scaled the initial implementation of the particle filter algorithm on CPU with different resolutions and particles.
CHAPTER 3

Related Work

Many of the previous research efforts focused on improving the execution speed of the particle filter algorithm by parallelizing the algorithm to take benefit of different types of parallel hardware platforms. In addition, previous research efforts focused on algorithmic techniques to accelerate the algorithm by optimizing the Bayes approximation. An example in this algorithm is the study in [16], which used variational Bayes approximation [17,18] as one-step approximation to draw necessary moments from the $N_s$ particles, which can yield a single-component marginalized filtering distribution. The simulations show the equivalent approach provided the best performance as an approximation of the posterior pdf at a shorter execution time.

Next, we provide a survey the most relevant previous studies that improve the execution time of the particle filter algorithm through parallelization approaches to take benefit of dedicated parallel hardware. A first type of such hardware is the FPGA. For example, the study in [19] improves the execution time of the particle filter algorithm in tracking applications by implementing parallel pipelining versions of the algorithm to be executed on FPGAs. More specifically, they investigated two techniques. In the first technique, consecutive loops inside the algorithm are merged, which in turn helps to reduce the execution time. In the second technique, sequential loops are implemented in a pipeline fashion to improve the throughput and execution time. The simulation results showed the fact that the optimized implementation was up to 11x faster
than the unoptimized one, for 1,000 particles. However, their experiments were not conducted in the context of tracking objects in videos. Instead, they used the particle filter algorithm to estimate the state of a complex system described by a state space model.

The study in [20] proposed a Metropolis coupled Markov Chain Monte Carlo (MC)3 approach [21] to address the problem of long computational time of the particle filter executed on a hardware-software platform. The hardware is based on the COordinate Rotation DIgital Computer (CORDIC) and random number generator (RNG). The software portion is executed on the Microblaze embedded software processor. They reported a speed-up of 3.97x compared to serial single hardware and software implementations and 72.28x compared to a software implementation.

The study in [22] proposed a parallel implementation of a histogram-based particle filter for object tracking on smart cameras based on single instruction multiple data (SIMD) processors. First, their approach extracts the relevant image features and store them in the external memory and then they are reorganized into the line memory in SIMD processor. Then the parallel computation of color histograms is processed by computing each row of particle regions in parallel. In their performance analysis, they tested the implementation on a desktop computer with five video sequences from the PET 2001 data set. They used 320 particles. In the HOG-based particle filter, their method outperforms the standard implementation when the particles are more than 175 particles. In color-based tracker, their method can achieve at most 30 images per
second. Their studies shows that it is possible to port the histogram-based particle filter to smart camera based on SIMD processors.

Although many of these FPGA-based implementations provide significant improvements, they suffer from not being very portable. In other words, these implementations can be executed out of the box only on the specific FPGAs that they are targeted for. In this thesis, our focus is however on parallelization of the particle filter algorithm using CUDA which has the advantages of being more platform independent and easier to learn and more accessible to a larger number of programmers, compared to FPGA approaches that require very specialized VHDL or Verilog programming skills. Therefore, we next describe previous studies that also used CUDA programming to speed up the particle filter algorithm.

The study in [23] proposed a CUDA implementation of visual tracking by applying the particle filter algorithm with pixel ratio likelihood that was executed on GPGPU. Their implementation can be divided into two parts. In the first part, they implemented colored-object tracking in a general scene, where the object appearance is unknown, and it may vary in time. This approach evaluates the likelihood of the rectangle target region based on its color. In the second part, they implemented hands tracking of a car driver [24,25]. It is based on a pixel ratio likelihood that was applied with the different observation model compared to the object tracking in a general scene. In this model, the likelihood for hands tracking includes two parts: left hand and right hand. These two likelihood functions form the basis of the weights of the particles. Their
implementations were tested on several GPUs platform individually GeForce GTX285, GTX675M, and Tesla C1060. The performance of the algorithm achieved 30 fps, for 8192 particles. The hands tracking of a car driver achieved more than 10 fps, for 8192 particles.

The study in [26] proposed a CUDA implementation of the particle filter algorithm that used appearance-adaptive models [27] executed on GPU. The prediction step of the particles is parallelized in two kernels using the Mersenne Twister kernel [28] provided by the CUDA™ SDK. Each thread generates two random numbers. The calculation of the particle weights is processed by two kernels using appearance-adaptive models to calculate the likelihood of the object. The size of the reference object template is 42 x 32 pixels. Each thread processes one column of the template. In the normalization step of the particle weights, the calculation is done with the use of the parallel reduction [29]. They reported that the execution time of the parallelized particle filter algorithm implementation achieved a speed-up of 30x compared to the CPU implementation, for 512 particles and for a 96 x 64 video resolution. In their experiments, the communication delays for transferring images from CPU to GPU have not been taken into account. The delay time of transferring data may have a significant increase on the execution time of their CUDA implementation and decrease the overall performance improvements.

The study in [30] proposed a CUDA implementation of the particle filter algorithm for pedestrian detection and tracking at night-time. CUDA programming is used to speed up the execution time of this implementation. The
pedestrian detection is performed by the Adaboost algorithm based on Haar-like features. The pedestrian tracking is processed by the particle filter algorithm on HSV histogram features. In their experiments, the dataset was from the videos of the Korea Internet and Security Agency. The videos are downsized to 480 x 320 video resolutions. The authors reported that the processing speed of the GPU implementation is 6.4x faster compared to the CPU implementation.

The study in [31] proposed the parallelization of the particle filter algorithm in a single target video tracking application. Multiple styles of parallel programming were used to increase the efficiency of the particle filter algorithm. First, the authors implemented the algorithm in MATLAB. Then, the code in MATLAB was translated to C line by line. The next step after the C implementation is to parallelize the Update step portion of the algorithm using OpenMP. There are two parts in their CUDA implementations of the particle filter algorithm. First, they parallelized the Update step of the algorithm using CUDA. Second, they parallelized the algorithm with a full CUDA implementation. In their experiment, the input was a synthetic video sequence. The resolution of the video was 128 x 128 pixels. They reported that the full CUDA implementation was not faster than the OpenMP implementation until around 9,000 particles are used. The CUDA implementations achieved the maximum speed-up of 32x compared to C implementation for 100,000 particles for 10 frames. The corresponding execution time of the CUDA implementation was 252 milliseconds per frame.

The study in [32] presented a parallelized implementation of a
decentralized particle filter for multiple objects tracking. The authors investigated two variants of the implementation. In the first variant, they performed multiple objects tracking by assigning one tracker to one object. The trackers were executed sequentially on GPU. Another variant is to group memory transfers and kernel executions together. In their experiments, several video datasets were tested. They reported the average speed-ups of 5x for 1,000 particles.

Tables 3.1 and 3.2 provide a comparison of the proposed work to these previous studies. From the tables, we can see that most of the previous work executed on GPUs used low-resolution videos to test the performance of the CUDA based implementation. For example, the study in [26] used 96 x 64 videos as input. As a result, the execution time for each frame in [26] is only 7.51ms. However, in real tracking applications, most of the cameras use at least 640 x 480 resolution. In other words, 96 x 64 video resolution is not feasible for real-world tracking application. In this thesis, we investigate video resolutions from 640 x 360 to 3840 x 2160 pixels to show the possible performance of the parallelization of the particle filter algorithm. In addition, although some previous studies had dramatic speed-up, they used a large number of particles in their experiment. This is because as the number of the particles increase, the execution time of the particle filter algorithm on CPU increases dramatically. The dramatic speed-up is obtained by apply a large number of particles to the algorithm. For example, in [31], 10,000 particles were used to track the object. The study in [30] is the closest to the proposed work in this thesis. It is based on the HSV histogram to update the weights of the particles. This model is also used in this thesis to
update the weights of the particles. However, in their study, the details of the parallelized implementation were not described, and the platform is unknown. The speed-up of the CUDA based implementation in this thesis is 7.5x, which is better than theirs which is 6.4x. The study in [32] used a testing platform that is the closet to the one in this thesis. The proposed work in this thesis achieved a speed-up of 7.5x, which is better than the approach in their study that is 5x speed-up.

While many of these parallelized implementations can perform dramatically better than their sequential counterparts, many previous studies do not specify details about their configuration of the particle filter beyond simple parameters. Because of that, it is hard to draw a conclusion as to what is the best parallelization of the particle filter algorithm among these previous studies. Also, most of the previous studies do not provide a comprehensive demonstration of the performance of the CUDA based implementation. Different from the previous studies, in this thesis, the detailed configuration of experiments on CUDA is given to explore the performance of the CUDA based implementation of the particle filter algorithm. Not only a series of prerecorded videos with different resolutions have been tested to investigate the performance of the particle filter using CUDA programming, but also the real-time video stream from a web camera has been tested with the parallelized algorithm to investigate the potential of the parallelized particle filter algorithm in real-time tracking applications. Furthermore, the detailed CUDA based programming is provided as an open-source code repository to present a clear view of the parallelization of the particle filter algorithm.
Table 3.1: Comparison of the proposed work to previous studies [Part1]
Approaches
Proposed Work
Full CUDA Implementation of
the Particle Filter [31]
Parallelization of
Multiple Objects Tracking [32]

Application
Object Tracking in Videos
Single Object Tracking
Multiple Objects Tracking

Input
Prerecorded Videos,
Real-time Streams
(360P - 2K Video Resolutions)
Synthetic Videos
(128 x 128 Video Resolution,
for 10 Frames)
Prerecorded Videos
(768 x 576 Video Resolution)

Number of Particles
1024 - 9216
100,000
500

CPU
3.5GHz Intel Xeon CPU 8 Cores
Intel Core i7
Intel Xeon E3-1241@3.5GHz

GPU
745MHz Nvidia Tesla K40C
2880 CUDA Cores
Nvidia GTX285
NVidia Quadro K420

Runtime
31ms - 448ms
252ms
500ms

Reported Speed-up
7.5x
32x
5x

Table 3.2: Comparison of the proposed work to previous studies [Part2]
3.1 Summary

In this chapter, a survey about the previous studies on the parallelization of the particle filter algorithm for tracking application is provided. The most relevant previous studies are discussed in more details in this chapter to help understand the difference between the proposed work and the others. In the next chapter, an introduction of parallel processing technologies will be provided.
CHAPTER 4

Parallel Processing Technologies

In this chapter, we provide a discussion of parallel processing technologies which will help understand the implementation method presented later in this thesis. A detailed illustration of the concept of CUDA programming is provided.

4.1 Basics of Parallel Computing

4.1.1 Serial Computing

Traditionally, programs have been written for serial execution. This meant that problems were solved by dividing them into a discrete series of instructions [33]. These discrete instructions are then executed on the CPU of a computer one by one. Only one instruction may be executed at any moment in time. After one instruction is done, the next one starts. Fig. 4.1 describes the serial computing. A real analogy of this would be people standing in a queue waiting for ordering food, and there is the only cashier serving one customer by one customer.

4.1.2 Parallel Computing

Parallel computing solves a computational problem by using computing resources simultaneously [33]. This means that the problems are broken into discrete parts that can be solved and executed concurrently. Each part is further broken into a series of instructions. Instructions execute simultaneously on
A common method of parallel computing is multithreading. The tasks are split up and run on different threads on the CPU, which decreases the total execution time. This is because multithreading uses the potential ability of each processor. Fig. 4.2 describes the parallel computing process.
thread and executes the tasks of the program simultaneously on threads.

Multithreading shows its significant advantage in the case that the tasks have the same runtime and their processes do not depend on each other. However, in the case of the tasks which are dependent relations, for example one task needs the result obtained from another task, in this case, the speed-up does not present significantly. It is relatively simple to accomplish multithreading on CPU compared to other parallel methods, such as CUDA programming, which is described next in this chapter.

4.1.3 GPGPU

When we talk about GPU, we usually think of applications and programs that are graphical interfaces of some sort [34]. For example, utilizing GPU is one of the most popular ways to use the graphical power of a GPU. A GPGPU is the use of not only graphics but also to handle computational tasks as CPU does. Specially, GPUs outperform most CPUs when handling the computational tasks that are highly parallelized. GPGPU, as a kind of parallel processing, allows programs to access the GPU alongside the CPU to speed up the overall performance of the computing tasks. GPGPU increases the efficiency by transferring some operations from CPU to GPU. Since GPUs are optimized to process vector or matrix calculations, they can be even more efficient than CPUs when processing some instructions [35]. These characteristics make a GPGPU an ideal choice to deal with computational problems. GPGPU is done by using programming languages that allow the CPU and GPU cooperate in operation system. The most popular such technique is CUDA programming.
4.2 CUDA Programming

In 2006, NVIDIA introduced CUDA, a general-purpose parallel computing platform, and a programming model for GPUs, which can help solve many complex computational problems more efficiently compared to CPUs. CUDA provides developers a standard programming language such as C to utilize the cores on GPU [36]. Usually, the more cores a GPU has, the more efficiently it can reach. In this thesis, we use NVIDIA Tesla K40c, which has 2880 CUDA cores.

The CUDA programming model is a heterogeneous model in which both CPU and GPU are utilized. In CUDA, the host refers to the CPU and its memory, and the device refers to the GPU and its memory. Code executed on the host is capable of managing memory on both the host and the device and also launch kernels, which are special functions executed on the device. These kernels are executed by many GPU threads in parallel. A typical sequence of operations is given below:

- Declare and allocate host and device memory.
- Initialize host data.
- Transfer data from the host to the device.
- Launch kernels.
- Transfer result from device to the host.

A simple example of vector addition using CUDA programming is presented below in Fig. 4.3. The detailed illustration of this example is given below.
// Device code
-_-global_-_ void VecAdd(float * A, float * B, float * C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float * h_A = (float *) malloc(size);
    float * h_B = (float *) malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float * d_A;
    cudaMalloc(&d_A, size);
    float * d_B;
    cudaMalloc(&d_B, size);
    float * d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}

Figure 4.3: CUDA example of vector adding [1].
4.2.1 Host Code

The function VecAdd is the kernel that executes on the GPU, and the main function is the host code. The main function declares three vectors A, B, and C. The pointers h_A and h_B point to the host vectors, allocated with malloc, and the pointers d_A, d_B, and d_C points to device vectors allocated with the cudaMalloc function from the CUDA runtime API. The host and device in CUDA have separate memory spaces, both of which can be managed from the host code. In order to initialize the device vectors, the data is copied from h_A and h_B to the corresponding device vectors d_A and d_B using cudaMemcpy, which works like the standard C memcpy function, except that it takes an argument which specifies the direction of the copy. cudaMemcpyHostToDevice is used to specify that.

In order to get the result back to the host, after running the kernel the data is copied from the device vector pointed to by d_C to the host vector pointed to by h_C by using cudaMemcpy and cudaMemcpyDeviceToHost.

The VecAdd is launched by the function in line 38 Fig. 4.3. The arguments in this function indicate the execution configuration, which describes the number of threads and blocks executed int this kernel. The CUDA thread is the basic unit of execution in a two-level hierarchy: blocks and grids. Threads exchange data through shared memory and global memory, respectively, in the same block or different blocks. Threads blocks and grids can have one, two, or three dimensions. After the kernel is finished, it is necessary to free any allocated memory, which is shown in line 44 Fig. 4.3. For device, cudaFree() is called.
Figure 4.4: Device code.

4.2.2 Device Code

In CUDA, the kernels described in Fig. 4.4 such as `VecAdd` are defined by using `__global__` declaration specifier. Variables defined within device code do not need to be specified as device variables because they are assumed to reside on the device. The `i` and `N` variables are stored as local memory by each in a register, and the pointers `A`, `B`, and `C` are the pointers stored as global memory to the device memory address space. The kernel is executed by multiple threads in parallel. If we want each thread to process an element of the vector, we need to specify an index to identify each thread. In CUDA, the variables `blockDim`, `blockIdx`, and `threadIdx` are the predefined variables which are of type `dim3`. The variables `blockDim` indicates the dimensions of each thread block as specified in the second execution configuration parameter for launching the kernel. The variables `blockIdx`, and `threadIdx` indicate the index of the thread within its block and the block within its grid, respectively. The global index `i` is used to access the elements of vector. In each thread, the addition of vector `A` and `B` is processed. The CUDA C compiler, `nvcc`, is part of the NVIDIA CUDA Toolkit.
4.3 Summary

In this chapter, an introduction of parallel programming using CUDA is provided. This will help to understand the key method of parallel programming technology employed in this thesis. A detailed description of CUDA programming is given by using an example of vectors addition. In the next chapter, the detailed CUDA based implementation of the particle filter algorithm is described.
CHAPTER 5

CUDA-Based Implementation of the Particle Filter

This chapter describes the CUDA implementation proposed in this thesis. The description will follow the flowchart of the implementations from Fig. 2.4 to illustrate how the parallel program executes on both CPU and GPU to exploit the GPU’s parallelism towards better execution time.

5.1 The Flowchart of the CUDA-Based Implementation of the Particle Filter Algorithm

Implementing an algorithm to execute on a GPU is more complicated than only doing it to execute on a CPU. The programmer must have a good understanding of achievable parallelization degree and of data locality, which are the most critical design aspects for parallelization with CUDA. That is because the delay penalty for global data accessing is relatively high and may overwhelm the gains obtained by parallelization. Hence, in this thesis, the initialization step of particles for the first frame of the video is processed on the host. The task of the initialization step is to generate all the particles in the center of the ROI. All the particles have the same location information with zero weight. In fact, it is not worthy of parallelizing the initialization on GPU. This is because the computation of the initialization step is relatively small compared to the likelihood function, which accounts for more than 80% of the execution time of the sequential CPU implementation.

After the initialization step is done, the algorithm moves to the transition
step. In this step, the predicted particles are generated from a Gaussian
distribution on the host first. It is after the initialization step that the likelihood
function is called. This function is parallelized using CUDA in this thesis. The
same memory size of particles is allocated on the device, and then the particles
are copied from the host to the device. Once we have the same particles on the
device, the kernel function of the likelihood calculation is launched. On the
device, the calculation of the likelihood function for each particle is executed by
separate threads. Specifically, in each thread, the ROI is converted from RGB
color channels to HSV channels. In order to compare the Bhattacharyya distance
between predicted particles and initial particles to decide the weights for each
predicted particle, the histogram of the Hue channel is computed. After the
calculation of the color feature, the Hue histogram is normalized within the (0,1)
range. This is because after normalization, the Hue histogram is easily used to
compare the Bhattacharyya distance between the particles. Once all the threads
are synchronized to finish the processing of the kernel function. The particles are
copied back to the host, which are used by the CPU for the next computation
described in Normalization and Resampling steps described in Chapter 2. The
flowchart of CUDA-based implementation of the particle filter algorithm is shown
in Fig. 5.1.

5.2 Detailed Description of The CUDA-based Implementation

Next, we will provide the detailed CUDA-based implementation of the
particle filter algorithm, including the preparatory work to transferring the data
and the CUDA kernel that computes the likelihood function for particles.
5.2.1 Preparatory Work

As described in Chapter 3, the data stored in the system RAM is not accessible by the device. The same data set of particles should be created on the device first, before launching any kernels on the device. However, in the sequential design, the particles are stored in the \texttt{dogs} vector, and copying this vector data back and forth from the system RAM to the device global memory is not feasible. The vectors have to be transferred to the arrays first on the host, and then the arrays are to be copied to the device memory for parallel
Figure 5.2: Function that initializes DogArr and prepares to store particles information obtained from Dog Vector.

computing. In Fig. 5.2, The `DogArr::iniDogArr` function creates a series of arrays that will store the data of particles obtained from vectors on the host.

Furthermore, the arrays have the same size as the number of the particles. The `DogArr::init_iDogArr` function assigns the same histogram feature to each particle that is stored in an array `hist`. This is because in the initialization session, all the particles are located in the center of the ROI, and therefore they have the same color histograms.

Once we have the information of all the particles, the process of copying data to the global memory on GPU can be started. This is described in Fig. 5.3.

The `DogArr::iniDogArr_d` allocates the memory space for the particles on
```c
void DogArr::iniDogArr_d(int num, DogArr *tDogArr)
{
    CUDAError_t mStatus;
    n = num;
    is_GPU = true;
    CUDAMalloc(&fw, n * sizeof(int));
    CUDAMalloc(&fh, n * sizeof(int));
    CUDAMalloc(&x0, n * sizeof(float));
    CUDAMalloc(&y0, n * sizeof(float));
    CUDAMalloc(&xp, n * sizeof(float));
    CUDAMalloc(&yp, n * sizeof(float));
    CUDAMalloc(&x, n * sizeof(float));
    CUDAMalloc(&y, n * sizeof(float));
    CUDAMalloc(&sp, n * sizeof(float));
    CUDAMalloc(&s, n * sizeof(float));
    CUDAMalloc(&width, n * sizeof(float));
    CUDAMalloc(&height, n * sizeof(float));
    CUDAMalloc(&weight, n * sizeof(float));
    mStatus = CUDAMalloc(&hist_d, 256 * sizeof(float));
    if (mStatus != CUDASuccess)
    {
        printf("Error 1");
        return;
    }
}
```

Figure 5.3: Function to allocate memory for the particles on GPU.

the device using CUDAMalloc, which is similar to the way memory is allocated on the device in the simple CUDA example described in Fig. ???. As shown, using CUDAError_t allows for the programmer to check the errors while compiling the program. All of these operations are done during the Initialization session of the CUDA-based implementation. The Initialization only happens on the first frame of the prerecorded video or real-time stream. After that, the algorithm moves to the Transition step, which is done for every frame except the first frame.

In the Transition step, the initial particles are first updated to the predicted particles by applying a second-order, auto-regressive dynamical model
void DogArr::fresh_iDog(int i, Dog *idog)
{
    fw[i] = (*idog).fw;
    fh[i] = (*idog).fh;
    x0[i] = (*idog).x0;
    y0[i] = (*idog).y0;
    xp[i] = (*idog).xp;
    yp[i] = (*idog).yp;
    x[i] = (*idog).x;
    y[i] = (*idog).y;
    sp[i] = (*idog).sp;
    s[i] = (*idog).s;
    width[i] = (*idog).width;
    height[i] = (*idog).height;
    weight[i] = (*idog).weight;
}

Figure 5.4: Function to fill in the arrays with the predicted particles.

given in equation 2.4. This step is processed on the host. These predicted particles are copied to the arrays on the host and then they fill in the memory space which is already allocated on the device, as shown in Fig. 5.4 and Fig. 5.5.

The DogArr::fresh_iDog and DogArr::freshDogArr_d functions are executed only once for each new frame in the video stream. These two functions ensure that on the device global memory, for every frame, the predicted particles always come from the step of the generation of predicted particles which is done on the host. Then, on the device kernel functions, these predicted particles is used to calculate the new color histograms for predicted particles and decide their weights. Before jumping into the detailed description of CUDA kernels, there is one remaining problem to be discussed. That is the fact that in OpenCV, the images are usually stored in cv::Mat which is a basic image container with two data parts [37]: the matrix header (containing information such as the size of the matrix, the method used for storing image, at which address is the matrix stored, and so on) and a pointer to the matrix containing
```c
void DogArr::freshDogArr_d(int num, DogArr *tDogArr)
{
    CUDAError_t mStatus;
    n = num;
    is_GPU = true;
    CUDAMemcpy(x, tDogArr->x, n * sizeof(float),
               CUDAMemcpyHostToDevice);
    CUDAMemcpy(y, tDogArr->y, n * sizeof(float),
               CUDAMemcpyHostToDevice);
    CUDAMemcpy(s, tDogArr->s, n * sizeof(float),
               CUDAMemcpyHostToDevice);
    CUDAMemcpy(width, tDogArr->width, n * sizeof(float),
               CUDAMemcpyHostToDevice);
    CUDAMemcpy(height, tDogArr->height, n * sizeof(float),
               CUDAMemcpyHostToDevice);
    mStatus = CUDAMemcpy(hist_d, tDogArr->hist, 256 * sizeof(float),
                         CUDAMemcpyHostToDevice);
    if (mStatus != CUDASuccess)
    {
        printf("Error 5");
        return;
    }
}
```

Figure 5.5: Function that copies predicted particles from host RAM to device global memory.

The pixel values. Not only the particles need to be transferred to the device, but also the images. In order to transfer image values to the host, the matrix `mat` is copied to an array. In Fig. 5.6, `Police_d::mat2float` function transfers the pixels of an image to an array `f`. Then, `at<uchar3>` is used to visit the element of the image matrix, where `uchar3` indicates the type and length of the element in the matrix. In the case of `uchar3`, the type of the element is `unsigned char`, and its length is 3, which correspond to the blue, green, and red channels for a single pixel in an image. Fig. 5.7 illustrates the processes of allocation and memory copying of images to the device. At this point, all the necessary data from the host for parallel computation of the likelihood function is on the device global memory.
```
void Police_d::mat2float(cv::Mat &mat, float *f)
{
    for (int i = 0; i < mat.rows; i++)
    {
        for (int j = 0; j < mat.cols; j++)
        {
            uchar3 v = mat.at<uchar3>(i, j);
            f[(i * mat.cols + j) * 3 + 0] = (float)v.x;
            f[(i * mat.cols + j) * 3 + 1] = (float)v.y;
            f[(i * mat.cols + j) * 3 + 2] = (float)v.z;
        }
    }
}
```

Figure 5.6: Function to copy images to the array.

```
CUDAError_t mStatus;
if (f==NULL)
{
    f = new float[frame.rows*frame.cols * 3];
    CUDAAlloc(&f_d, frame.rows*frame.cols * 3 * sizeof(float));
    mat2float(frame, f);
    mStatus = cudaMemcpy(f_d, f, frame.rows*frame.cols * 3 * sizeof(float), cudaMemcpyHostToDevice);
    if (mStatus != CUDASuccess)
    {
        printf("Error 5");
        return;
    }
}
```

Figure 5.7: Function to transfer the images to device.

### 5.2.2 Kernel Implementation

The CUDA kernel functions parallelize the likelihood function on the device. The likelihood function is responsible for the calculation of Hue histograms for the particles and for the comparison of their histograms to finally evaluate the estimation of the particles and to decide their weights. The theoretical equation of this step is equation 2.15 in Chapter 2. The likelihood function of the original implementation executed on CPU is named
```cpp
float Dog::_likelihood(const cv::Mat &frame) {
    int c = cvRound(this->x);
    int r = cvRound(this->y);
    int w = cvRound(this->width * this->s);
    int h = cvRound(this->height * this->s);
    cv::Mat imgROI = frame(cv::Rect(c - w / 2, r - h / 2, w, h));
    cv::Mat *hist = pHist->getHueHistogram(imgROI, 40);
    float d_sq = 1.0f * cv::compareHist(*this->hist, *hist,
                                         CV_COMP_BHATTACHARYYA);
    return std::exp(100 * d_sq);
}
```

Figure 5.8: Function that calculates Hue histogram for each predicted particle and assigns the weight based on its Bhattacharyya distance with respect to the original particle.

`_likelihood` and given in Fig. 5.8 to provide a clear understanding of its operation and a comparison compared to its CUDA version on the device. In the CUDA-based implementation, the kernel, shown in Fig. 5.9, is very simple. In this figure, `ith` indicates the index for thread calculation based on the `blockIdx`, `blockDim`, and `threadIdx`, which are discussed in Chapter 3.

The `_likelihood_d` function is the CUDA version of the `_likelihood` on the device. The result of `_likelihood_d` will be saved to the weight array after running on a particular index.

In this thesis, we program using CUDA the `_likelihood_d` function to replace it with a kernel function, which will be executed on the device. The source code of `_likelihood_d` is shown in Fig. 5.10. The `_device_` specifier indicates that the device functions can only be called by other device or global functions and cannot be called by host code. It is executed only on the device.

If we compare Fig. 5.8 and 5.10, we can see that the CUDA `_likelihood_d` function performs the exact same operations as the CPU
-global-_ void dog_Run_d(int threadNum, int n, float *frame, int rows, int cols,
  float *x, float *y, float *width, float *height, float *s,
  float *hist, float *weight
)
{
  const int i = blockIdx.x;
  const int j = threadIdx.x;

  int ith = i*threadNum + j;
  if (ith >= n)
  {
    return;
  }

  weight[ith] = _likelihood_d(ith, frame, rows, cols, x, y,
    width, height, s, hist, weight);
}

Figure 5.9: CUDA kernel implementation.

function. Specifically, first the image ROI (i.e., imgROI) is drawn by two loops,
as described in lines 4 to 31 in Fig. 5.10. Next, the imgROI is used in the
calculations of the Hue histogram. The programming of this part is significantly
more difficult compared to other portions of the code. This is because OpenCV,
as an open-source library for computer vision, provides so many built-in
functions for programmers. Some of them are not available for GPU devices. The
getHueHistogram function of CPU implementation in line 8 of Fig. 5.8 calls
several built-in functions from the OpenCV library, which contribute to the
calculations of the color histogram, as described in Fig. 5.11. In that figure,
cv::cvtColor, cv::calcHist, and cv::normalize are utilized to
compute the Hue histogram for each particle. In order to be able to execute these
functions on the device, we re-write these functions.

On the device, the ColorHistogram_d::getHueHistogram function
shown in Fig. 5.12 covers the operations done by cv::cvtColor,
```c
#define _device_
float _likelihood_d(int ith, float *frame, int rows,
    int cols,
    float *x, float *y, float *width, float *height, float *s,
    float *hist, float * weight)
{
    int c = round(x[ith]);
    int r = round(y[ith]);
    int wCol = round(width[ith] * s[ith]);
    int hRow = round(height[ith] * s[ith]);

    float imgROI[maxRow*maxRow * 3];

    if (c + wCol / 2 > cols) c = cols - wCol / 2;
    if (c - wCol / 2 < 0) c = wCol / 2;
    if (r + hRow / 2 > rows) r = rows - hRow / 2;
    if (r - hRow / 2 < 0) r = hRow / 2;

    int iRow, iCol, iD;
    for (int i = 0; i < hRow; i++)
    {
        for (int j = 0; j < wCol; j++)
        {
            iCol = c - wCol / 2 + j;
            iRow = r - hRow / 2 + i;
            iD = (iRow*cols + iCol) * 3;

            { imgROI[(i*wCol + j) * 3 + 0] = frame[iD + 0];
              imgROI[(i*wCol + j) * 3 + 1] = frame[iD + 1];
              imgROI[(i*wCol + j) * 3 + 2] = frame[iD + 2];
            }
        }
    }

    ColorHistogram_d pHist2;
    ColorHistogram_d_Init(pHist2);

    float normal_hist0[256];
    ColorHistogram_d_getHueHistogram(pHist2, imgROI, hRow, wCol,
        40, normal_hist0);

    float distance = 1.0f - BhattacharyyaDistance(pHist2, hist,
        normal_hist0);
    distance = exp(100.0f * distance);
    return distance;
}
```

Figure 5.10: Likelihood function on device.

cv::calcHist, and cv::normalize. First, the bgr2hsv function shown in lines 1 to 30 of Fig. 5.13 applies the equation used to convert Blue, Green, and Red color space to Hue, Saturation, and Value color space. The BGR2HSV
cv::Mat *ColorHistogram::getHueHistogram(const cv::Mat &image, int minSaturation) {
    cv::Mat *hist = new cv::Mat();
    cv::Mat hsv;
    cv::cvtColor(image, hsv, CV_BGR2HSV);
    cv::Mat mask;
    if (minSaturation > 0) {
        std::vector<cv::Mat> v;
        cv::split(hsv, v);
        cv::threshold(v[1], mask, minSaturation, 255, cv::THRESH_BINARY);
    }
    cv::calcHist(&hsv, 1, channels, mask, *hist, 1, histSize, ranges);
    cv::Mat *normal_hist = new cv::Mat();
    cv::normalize(*hist, *normal_hist, 1.0);
    return normal_hist;
}

Figure 5.11: Function that calculates of Hue histogram on CPU.

function shown in lines 31 to 48 of Fig. 5.13 calls the bgr2hsv function that defines the operation for color space conversion and converts the imgROI from Blue, Green, and Red color space to Hue, Saturation, and Value color space. The OpenCV documentation provides the details about the color space conversion, as shown in equation 5.1. Next, once the HSV color space is available, the device function moves to the calculation of Hue histogram, which is shown in lines 5 to 53 of Fig. 5.12. The histogram is a very important feature for image processing, as a graphical representation of the distribution data [38]. A Hue histogram gives the distribution of the Hue value of pixel intensities in a digital image. In this thesis, the image pixel uses 8 bits, and so the range of Hue is 0 – 255.

Furthermore, we set the bin size to 256 so that it would be relatively easy to count the number of pixels that fall in the range of each bin because the Hue value of the pixel indicates its bin range. In order to compare the Bhattacharyya
distance between the particles easily, the result of the histogram is normalized to
the range (0, 1). The equation used in the *Normalization* step is given by the
equation 5.2.

\[
V = \max(R, G, B)
\]

\[
S = \begin{cases} 
(V - \min(R, G, B))/V, & V \neq 0 \\
0, & \text{Otherwise}
\end{cases}
\]

\[
H = \begin{cases} 
60(G - B)/(V - \min(R, G, B)), & V = R \\
120 + 60(B - R)/(V - \min(R, G, B)), & V = G \\
240 + 60(R - G)/(V - \min(R, G, B)), & V = B,
\end{cases}
\]

(5.1)

where \(H\) is the Hue channel, \(S\) is the Saturation channel, and \(V\) is the Value
channel. \(R, G, B\) are the Red, Green, Blue channels of the images.

\[
dst(i, j) = \frac{src(i, j)}{\sqrt{\sum src(i, j)^2}},
\]

(5.2)

where \(i, j\) are the index of the element in Hue Histogram and \(src\) is the Hue
histogram. The *ColorHistogram_d_getHueHistogram* function from Fig.
5.12 returns the normalized Hue histogram array, which is used to compare the
Bhattacharyya distance with the Hue histogram of the particle from last frame as
shown in Fig. 5.14. Their difference finally decides the weights of the predicted
particles. The Bhattacharyya distance is used to measure the overlap between
the two histograms. A small value result indicates a good match and a large
value result indicates a poor match. The equation for computing the Bhattacharyya distance is given in equation 5.3.

\[ d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{H_1 \cdot H_2 N^2}} \sum_I \sqrt{H_1(I)H_2(I)}}, \]

(5.3)

where \( H_1(I) \) and \( H_2(I) \) represent the elements of the Hue Histograms. \( \overline{H}_1 \) and \( \overline{H}_2 \) represent the mean of Hue Histograms.

Once the calculation of the Bhattacharyya distance is done, the CUDA _likelihood_d function returns the distance result to the weights array, as shown in line 14 of Fig. 5.9. This indicates the kernel function dog_Run_d has finished the task on the GPU device. CUDADeviceSynchronize() is used to synchronize all the threads to check the status whether all the current tasks are completed and returns CUDASuccess. After threads synchronization, the weights array is copied from the GPU memory to the host RAM memory to _dogArr. Then it is transferred to _dogs vector defined on CPU. CUDAFree() is called to free any allocated memory that defined on the host. For the host memory, it is free().

The implementation of _likelihood_d function on GPU is completed, when _dogs vector on CPU receives the new weight values for the predicted particles. Then, the algorithm continues to the Normalization and Resampling steps described in Fig. 5.1.
5.3 Summary

In this chapter, we presented the CUDA-based implementation of the likelihood calculation of each particle discussed in Chapter 2. The performance comparison and its application on real-time streams of the CUDA-based implementation will be investigated in Chapter 5.
```c
__device__ void ColorHistogram_d_getHueHistogram(ColorHistogram_d &data, float image[maxRow*maxRow*3], int rows, int cols, int minSaturation, float normal_hist0[256])
{
    BGR2HSV(image, rows, cols);
    int mask[maxRow*maxRow];
    {
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                int v = image[(i*cols + j) * 3 + 1];
                if (v > minSaturation)
                {
                    mask[i*cols + j] = 1;
                }
                else
                {
                    mask[i*cols + j] = 0;
                }
            }
        }
    }
    float max = 1e-10;
    
    for (int iH = data.ranges[0][0]; iH < data.ranges[0][1]; iH++)
    {
        normal_hist0[iH] = 0.0;
    }
    int v4 = 0;
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            v4 = (int)image[(i*cols + j) * 3 + 0];
            normal_hist0[v4] += mask[i*cols + j];
        }
    }
    
    for (int iH = data.ranges[0][0]; iH < data.ranges[0][1]; iH++)
    {
        max += normal_hist0[iH] * normal_hist0[iH];
    }
    max = sqrt(max);
    for (int iH = data.ranges[0][0]; iH < data.ranges[0][1]; iH++)
    {
        normal_hist0[iH] /= max;
    }
    return;
}
```

Figure 5.12: Function to calculate of the Hue histogram on device.
```c
_device_ int bgr2hsv(float b, float g, float r, float &h, float &s, float &v)
{
  b /= 255;
  g /= 255;
  r /= 255;

  float max = b;
  if (g > max) max = g;
  if (r > max) max = r;

  float min = b;
  if (g < min) min = g;
  if (r < min) min = r;

  if (r == max)
    h = (g - b) / (max - min);
  if (g == max)
    h = 2.0 + (b - r) / (max - min);
  if (b == max)
    h = 4.0 + (r - g) / (max - min);
  if (h < 0)
    h = h + 360.0;
  v = max;

  s = (v - min) / v;
  if (v == 0) s = 0;
  return 0;
}

device_ void BGR2HSV(float image[maxRow*maxRow * 3], int rows, int cols)
{
  float b; float g; float r; float h; float s; float v;
  for (int i = 0; i < rows; i++)
  {
    for (int j = 0; j < cols; j++)
    {
      h = image[(i*cols + j) * 3 + 0];
      g = image[(i*cols + j) * 3 + 1];
      r = image[(i*cols + j) * 3 + 2];
      bgr2hsv(b, g, r, h, s, v);
      image[(i*cols + j) * 3 + 0] = ((h / 2.0));
      image[(i*cols + j) * 3 + 1] = ((s * 255));
      image[(i*cols + j) * 3 + 2] = ((v * 255));
    }
  }
  return;
}
```

Figure 5.13: Functions to convert RGB to HSV on device.
Figure 5.14: Function that calculates of Hue Bhattacharyya distance between the predicted particles.
CHAPTER 6

Discussion of Results

In this chapter, we conduct several experiments to investigate the performance of the CUDA-based implementation of the particle filter algorithm provided in the previous chapter. The results are compared to those obtained using the sequential CPU version of the same algorithm.

Each of the tests were performed three times, and the results were averaged before being included in each table. Four experiments were performed:

• 1) Scalability with image resolution. This experiment tests the performance of the CUDA-based implementation for different image sizes. When the number of pixels in each frame increases, the execution time of the CUDA-based implementation is expected to be significantly shorter than that of the CPU version.

• 2) Scalability with number of particles. When the number of particles increases, the speed-up of the particle filter algorithm is expected to be far better because the processing operations are parallelized for all particles.

• 3) Applying the CUDA implementation to a dynamic or real-time video stream from an actual web camera. In this thesis, the manual selection of the initialization of the particle filter algorithm is replaced with automatic detection. More specifically, face detection is applied to detect the human faces and initialize the ROI. The selected faces are expected to be tracked by using the particle filter algorithm.
4) To evaluate the performance of CUDA-based implementation, we perform the experiments on a video sequences Bolt data set from OTB100 benchmark. A breakdown of the execution times of main stages is shown next to help understand how the parallel implementation benefits from CUDA programming.

All tests were performed on a system with the following specifications:

- 3.5 GHz Intel Xeon CPU 8 Cores
- 745 MHz Nvidia Tesla K40C GPU
- PCIe version 3.0
- Ubuntu 18.04 LTS
- Nvidia driver version 390.116
- CUDA Toolkit version 9.1
- OpenCV library 3.4.2

6.1 Results from Testing Different Video Frame Sizes

In order to test the performance on the CUDA version for different image resolutions, a pre-recorded video with varying resolutions from 640 x 360 pixels to 3840 x 2160 pixels, was compiled and tested. This video shows a tennis ball rolling on the floor from left to right. The original video was recorded with the 3840 x 2160 resolution and then downsampled to other resolutions listed above to test the performance for different resolutions. Each implementation was able to
Figure 6.1: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 1024 particles.

track the ball. As we expected, the execution times were different in each case.

The comparisons of all execution times of the CUDA and CPU implementations for different image resolutions are shown in Fig. 6.1, 6.2, 6.3, and 6.4. A summary of the speed-up is presented in Fig. 6.5.

It can be observed that the runtime of the CUDA-based implementation is shorter than the one on CPU for all configurations of video resolutions. This is because the likelihood function for all particles is executed as a parallelized function by multiple threads. The short execution time of the GPU implementation compared to the CPU implementation indicates that the parallelization has been successful in speeding up the execution time. In addition, we observed that when the video resolution increases, the total execution time increases for both implementations. This is because in the `likelihood` function, the amount of calculations of the Hue histogram is proportional to the
Figure 6.2: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 3072 particles.

Figure 6.3: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 6144 particles.

number of pixels in the frame. As a result, when the video resolution increases, the execution time of the main function will increase as well. A summary of all experiments is shown in Fig. 6.5. In the tests with the same number of the
Figure 6.4: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different video resolutions, for 9216 particles.

Figure 6.5: The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different video resolutions. particles, the speed-up does not change when the video resolution increases. The minimum speed-up is around 1.6x when the number of particles is 1024. The maximum speed-up is 7x when the number of the particles is 9216. This indicates
that the video resolution does not have a significant impact on the speed-up. The is because calculation for all the particles are parallelized to each thread and calculation of the Hue histogram for each particle is processed sequentially in each thread. The execution time of the calculation of the Hue histogram for a particle increases because of higher number of the pixels in a frame, hence, the runtime for both CUDA and CPU implementations increases proportionately.

6.2 Results for Different Numbers of Particles

In this experiment, we tested the CUDA implementation for a varying number of particles with respect to different video resolutions. The resulting execution times are shown in Fig. 6.6, 6.7, 6.8, 6.9, and 6.10. We observed that the CUDA implementation is the fastest in all the tests, as we expected. The execution time of the CPU implementation increases for all the video resolutions at a much faster rate as the number of particles increases, but the CUDA-based implementation maintains its significant advantage. This makes the CUDA implementation the preferred option when the particle filter algorithm is used to track objects with a very large number of particles in a high-resolution video.

Fig. 6.6 shows the variation of the execution time for both CUDA and CPU implementations with a constant resolution of 640 x 360. It can be seen that when the number of particles is small (e.g., 1024), the CUDA implementation has a performance similar to that of the CPU implementation. When the number of particles increases, the execution of CUDA implementation increases only slightly, which makes the CUDA implementation to maintain a
Figure 6.6: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 640 x 360 video resolution.

Figure 6.7: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 856 x 480 video resolution.

significant lead in the performance at large numbers of particles, at different video resolutions such as 856 x 480, 1280 x 720, 1920 x 1080, and 3840 x 2160 as shown in Figures 6.7 to 6.10.
Figure 6.8: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 1280 x 720 video resolution.

Figure 6.9: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 1920 x 1080 video resolution.

Fig. 6.11 shows the speed-up obtained by the CUDA implementation when processing a different number of particles. This figure indicates that the speed-up of the CUDA implementation is affected by the number of particles. No
Figure 6.10: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementation with respect to different numbers of particles, for 3840 x 2160 video resolution.

matter which resolution of the video is, the speed-up obtained by the CUDA implementation remains the same. Nevertheless, it needs to be noticed that the resolutions of the video actually influence the total execution time of the CUDA implementation. Higher resolution video takes longer to process the computation. The reason behind that is the computation for each particle is parallelized in the threads executed on GPU using CUDA programming.

Fig 6.12 shows snapshots of the tracking of a ball by the CUDA implementation on our prerecorded video dataset.
Figure 6.11: The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different numbers of the particles.

Figure 6.12: Selected frames that show tracking of a prerecorded video, using the CUDA implementation.
6.3 Results for the Real-time Video Streams in Dynamic Surveillance Scenario

In order to test the performance of the CUDA-based implementation in a dynamic surveillance scenario, we tested it on a real-time video stream obtained from an actual web camera. In the initialization step of the particle filter algorithm, the manual selection of the target is replaced by the face detection using Haar cascade classifier. Haar cascade classifier is a machine learning based approach in which a cascade function is trained with a set of input data. OpenCV includes such pre-trained classifiers for face, eyes, body, etc. Therefore, the pre-trained classifier `haarcascade_frontalface_alt.xml` was used to detect a human face in the following experiments.
Several frames from this experiment are shown in Fig. 6.13. It can be seen that the faces have been detected automatically in the video stream by a web camera without human intervention. When the faces move around in the screen, the faces can still be tracked successfully on the following frames. The resolution of the video stream is 640 x 480 pixels, which is a common resolution for a web camera. In this experiment, the CUDA-based implementation with face detection can achieve the average speed-up of 6.5x compared to the CPU implementation. The original FPS of the video stream by a web camera is 30 FPS. The performance of the CUDA-based implementation with face detection can achieve 20 FPS for 6144 particles.

6.4 Results for the Bolt Dataset

In this section, we test the performance of the CUDA-based implementation on video sequences from the Bolt dataset benchmark OTB100 [39]. In the experiments, we used a number of varying particle numbers from 1024 to 9216. The video resolution is 640 x 360 pixels. The tracking region is 12 x 17 pixels. The variation of the execution time is shown in Fig. 6.14. As we expected, the CUDA implementation is faster in all the cases. When the particles number increases, the execution time of the CPU implementation increase significantly. However, the execution time of the CUDA implementation increases at a much slower pace. This demonstrates the advantage of the CUDA-based implementation.

Fig. 6.15 shows the speed-up obtained by the CUDA implementation
Figure 6.14: Variation of the execution time of the particle filter algorithm using CUDA and CPU implementations with respect to different particle numbers, on the OTB100 benchmark.

Figure 6.15: The relative speed-up obtained by the CUDA-based implementation compared to the CPU implementation with respect to different numbers of the particles.

when processing with a different number of particles. As we expected, the achieved speed-up in most of tests is the same as the speed-ups reported in Fig. 6.11. When the particle number is 9216, the maximum speed-up is 8x.
Table 6.1 shows the execution time for each processing step of the particle filter algorithm for both CPU and GPU implementations. As we expected, the execution time of the Transition step on GPU version is much shorter in all the cases. This is because the likelihood calculation of particles in Transition step is parallelized. Note that, execution time of the CPU implementation of this portion increases at a faster rate as the number of particles increase. In addition, the Sort and Resample steps take 126 µs and are executed 350 times, which is approximately 10% of the total execution time. This portion could be parallelized in future work.
<table>
<thead>
<tr>
<th></th>
<th>1024</th>
<th>3072</th>
<th>6144</th>
<th>9216</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Initialization</td>
<td>0.446 ms</td>
<td>94.6 ms</td>
<td>0.734 ms</td>
<td>95.8 ms</td>
</tr>
<tr>
<td>Transition</td>
<td>30.35 ms</td>
<td>10.1 ms</td>
<td>71.15 ms</td>
<td>12.3 ms</td>
</tr>
<tr>
<td>Sort</td>
<td>52.42 µs</td>
<td>83.48 µs</td>
<td>117.29 µs</td>
<td>237.4 µs</td>
</tr>
<tr>
<td>Normalization</td>
<td>7.58 µs</td>
<td>11.75 µs</td>
<td>19.83 µs</td>
<td>25.7 µs</td>
</tr>
<tr>
<td>Resample</td>
<td>74.24 µs</td>
<td>117.95 µs</td>
<td>223.31 µs</td>
<td>345.4 µs</td>
</tr>
</tbody>
</table>

Table 6.1: Execution times of main steps of the algorithm.
Fig. 6.16 shows snapshots of the tracking result of our CUDA implementation on the **Bolt** dataset.

![Tracking results on the Bolt dataset from OTB100 benchmark.](image)

6.5 Further Observations

The CUDA-based implementation of the particle filter algorithm has a significant advantage to speed up the program. However, the Telsa K40C is still an old GPU architecture, which was released in 2013. Although K40C is an expensive GPU aimed at enterprise customers, it is only compute capability 3.5, meaning that it does not support many new features and performance improvements available in newer GPUs. For example, the New Geforce 20 series is developed by Nvidia and was released in 2018. It is based on the latest Turing architecture and supports compute capability 7.5 and CUDA 10.0, which contain a lot of new features for high-performance computing and artificial intelligence.
For example, compared to the last generation of the GPU architecture Pascal, the computing ability of the newest Turing architecture can achieve 1.4 x faster. Also, the new GDDR6 VRAM has a larger bandwidth and capability. These features will dramatically improve the performance of the CUDA-based implementation.

6.6 Summary

In this chapter, several experiments are conducted to investigate the performance of the CUDA-based implementation of the particle filter algorithm. We found that the CUDA-based implementation of the particle filter is better than the CPU implementation. This is because the CUDA-based implementation take the benefit of the parallel computing ability of GPUs. The CUDA base implementation was able to obtain the maximum 7.5x speed-up for a 3840 x 2160 video resolution, using 9216 particles, while tracking the a tennis ball. In addition, a read-time video stream from a web camera was added to the CUDA-based implementation to track human faces. From the results, we found that the CUDA implementation can detect and track the faces successfully and perform 6.5x faster compared to the CPU implementation on execution time.
CHAPTER 7

Conclusion and Future Work

7.1 Conclusions

The main objective of this thesis is to speed up the execution of the CPU implementation of the particle filter algorithm for tracking targets using CUDA programming. First, we profiled the CPU implementation of the particle filter algorithm to find out that the calculation of the likelihood function accounts for 80% of the execution time. Therefore, the objective of this thesis is to parallelize the likelihood function using CUDA programming. Several device functions were used to implement the operations of the likelihood function on the GPU, which includes the preparatory work of data, color space conversion, calculation of Hue histogram, computation of Bhattacharyya distance, and normalization of the weights for the particles. These tasks for each particle are programmed using CUDA and executed in parallel by CUDA threads on GPU.

In the experiments, the CUDA based implementation and CPU implementation were tested to compare the performance of the parallelization of the particle filter algorithm to the base case. It was confirmed that the particle filter algorithm benefits from the parallel implementation using CUDA programming, and the CUDA based implementation was significantly faster than the CPU version. The CUDA based implementation was able to achieve a maximum 7.5x speed-up for a 3840 x 2160 video resolution, and 9216 particles compared to the CPU implementation. In addition, the algorithms were tested on a real-time video stream recorded by a web camera, to detect human faces. The CUDA based
implementation was able to detect human faces on the screen and keep tracking human faces. Our work provides the programming basis for future work on the particle filter algorithm for tracking human faces in surveillance scenarios.

There are some advantages to the CUDA based implementation. The CUDA version is faster at all tasks, especially when dealing with a large number of particles. Compared to traditional serial programming on CPU, CUDA presented its substantial advantages when executing the algorithm that contained a large amount of repeated data processing, such as the calculation of the likelihood function for a large number of times. This characteristic makes CUDA advantageous especially in some specific applications. Also, the programming interface of CUDA is based on the standard C language with extensions, which makes CUDA easier to learn and adopt. One can start to write simple CUDA programs with the basic programming experience of C language. However, CUDA is a very different style of programming language and programming method. It is relatively new and has fewer resources available compared to other traditional languages (e.g., C, C++). New CUDA versions are frequently released with new features, and this makes it more challenging for beginners. However, CUDA has rapid development that brings more and more new features, which optimizes CUDA programming for future use.

### 7.2 Future Work

The work in this thesis can be expanded in several directions. First, we can process many image pixels in parallel. The calculation of each image pixel
can be parallelized in each thread. This could significantly increase the advantage of CUDA programming, especially in high-resolution videos. Also, the CUDA implementation can be compiled and executed on newer GPU architectures, such as Telsa V100, which is the latest GPU in the Telsa series containing 5120 CUDA cores and 900GB/sec memory bandwidth, and other new features for CUDA programming. These new features are expected to provide substantial improvements for the performance of the CUDA based implementation.

Second, the real-time video stream implemented in the CUDA design by a web camera is another direction to optimize. The current resolution of the video stream is 640 x 480. Compared to the mainstream specifications of typical surveillance systems, 640 x 480 video resolution is relatively low. In future work, the use of a high-resolution web camera could be added to the CUDA design. However, with more pixels in the frame, the processing speed for each frame will slow down. This issue could cause the CUDA based implementation not to meet the requirement for real-time video stream. However, it may be solved with the above optimized parallelization of the computation for each pixel.

Finally, the current CUDA based implementation of the particle filter algorithm only computes the Hue histogram for each particle. The accuracy of tracking objects will be improved if Saturation and Value could be added into the calculation of histogram feature for the particles. Future optimization could be to compute the many histogram bins in parallel. In future work, each particle computes three histogram bins. The parallelization of histogram bins in each thread could improve the tracking accuracy of the CUDA based implementation,
REFERENCES


